Versione 1.6.1 - 22/02/2021

Reti di Calcolatori T – Ingegneria Informatica

Corso tenuto da Antonio Corradi

APPUNTI DI TEORIA

PER L'ESAME ORALE

Appunti di Andrei Daniel Ivan, andreidaniel.ivan@studio.unibo.it

per l'anno accademico 2020/2021

(nel caso troviate errori grammaticali, errori concettuali, cose poco chiare o aggiornamento delle slide, mandatemi una mail e aggiornerò il file)

Indice

1. Integrazione e programmazione di sistemi **7**

1.1. Programmazione di sistema 7

1.2. Programmazione a microservizi 7

1.3. Architettura delle soluzioni 7

1.4. Semplicità degli algoritmi e del codice 8

2. Generalità, obiettivi e modelli di base **9**

2.1. Introduzione ai sistemi distribuiti 9

2.2. Processi 9

2.3. Modello Client/Server 9

2.4. Modelli asincroni – pull e push 10

2.5. Modello a delegazione 10

2.6. Modello ad eventi 10

2.7. Modello ad agenti multipli 10

2.8. Stato 11

2.9. Tipi di server 11

2.10. Sistemi di nomi 11

2.11. DNS – *Domain Name System* 12

3. Programmazione di rete e modello C/S con socket in Java **13**

3.1. Introduzione 13

3.2. Socket datagram 13

3.3. Socket multicast 13

3.4. Socket stream 13

3.5. Chiusura e opzioni per le socket 14

4. Programmazione di rete e modello C/S con socket in C **16**

4.1. Introduzione 16

4.2. Domini di comunicazione, nomi e primitive preliminari 16

4.3. Presentazione dei dati 17

4.4. Socket datagram 17

4.5. Socket stream 18

4.6. Chiusura delle socket 18

4.7. Opzioni per le primitive 19

4.8. Modalità asincrone e non bloccanti per le primitive 19

4.9. La primitiva select 20

5. Servizi applicativi standard in Internet **21**

5.1. Introduzione 21

5.2. Telnet – *Virtual Terminal Protocol* 21

5.3. NVT – *Network Virtual Terminal* 22
5.4. rlogin – *Remote Login* 22
5.5. FTP – *File Transfer Protocol* 23
5.6. Posta elettronica 23
5.7. SMTP – *Simple Mail Transfer Protocol* 24
5.8. Usenet news 24
6. OSI – *Open System Interconnection* 25
6.1. Introduzione 25
6.2. Architettura 25
6.3. Nomi ed entità 25
6.4. Protocolli e implementazioni 26
6.5. Formato dei messaggi 26
6.6. Modalità di connessione 26
6.7. Primitive 27
6.8. Livello network 27
6.9. Livello trasporto 27
6.10. Livello sessione 28
6.11. Livello presentazione 29
6.12. Livello applicazione 30
7. Java RMI – *Remote Method Invocation* 31
7.1. Introduzione 31
7.2. Architettura RMI 31
7.3. Interfacce ed implementazione 32
7.4. RMI registry 32
7.5. Stub e skeleton 32
7.6. Serializzazione e passaggio dei parametri 33
7.7. Livello di trasporto: concorrenza e comunicazione 33
7.8. Distribuzione delle classi (deployment) e class loading 33
7.9. Sicurezza in RMI 34
8. Sistemi RPC – *Remote Procedure Call* 36
8.1. Semantiche di comunicazione 36
8.2. Introduzione ai sistemi RPC 36
8.3. Tolleranza ai guasti 37
8.4. NCA – *Network Computing Architecture* 37
8.5. Passaggio dei parametri e trattamento delle eccezioni 38
8.6. IDL - *Interface Definition Language* 38
8.7. Sviluppo e fasi di supporto RPC 38
8.8. RPC binding e sistemi di nomi 39

8.9. RPC asincrone 39
8.10. Proprietà delle RPC 40
9. Implementazione RPC di SUN 41
9.1. Introduzione 41
9.2. Definizione del programma RPC 41
9.3. Confronto tra sviluppo locale e sviluppo remoto 41
9.4. Identificazione di RPC 42
9.5. Livelli di RPC 42
9.6. Omogeneità dei dati 43
9.7. Portmapper 43
9.8. Modalità asincrona batch 44
10. Sistemi di nomi 45
10.1. Introduzione 45
10.2. Sistemi di nomi 45
10.3. Componenti di un sistema di nomi 45
10.4. Directory X.500 46
10.5. Protocolli di directory e protocolli di discovery 47
11. TCP/IP: protocolli e scenari d'uso 48
11.1. Introduzione 48
11.2. ARP – *Address Resolution Protocol* 48
11.3. RARP – *Reverse Address Resolution Protocol* 48
11.4. DHCP – *Dynamic Host Configuration Protocol* 49
11.5. NAT – *Network Address Translation* 49
11.6. ICMP – *Internet Control Message Protocol* 49
11.7. UDP – *User Datagram Protocol* 50
11.8. ARQ - *Automatic Repeat Request* e Continuous Requests 50
11.9. TCP – *Transmission Control Protocol* 51
11.10. TCP: fase iniziale 51
11.11. TCP: fase di comunicazione 52
11.12. TCP: fase finale 53

Prefazione

Il seguente documento, scritto per facilitare lo studio per l'esame orale di Reti di Calcolatori T, non ha lo scopo di sostituire le slide e gli strumenti di studio forniti dal docente; chiunque lo usi per studiare si assume quindi la responsabilità di non star studiando dal materiale di studio fornito del professore ma da un documento scritto sulla base di esso, che può quindi presentare qualche errore. Tuttavia, ritengo che studiare da questo documento risulti molto più comodo in quanto le slide non sono impostate per essere in forma testuale, e quindi facilmente leggibili, ma sono più impostate per essere spiegate in aula.

1. Integrazione e programmazione di sistemi

1.1. Programmazione di sistema

Accanto alla programmazione applicativa, fatta per raggiungere obiettivi specifici e costituita da applicazioni che rispondono a esigenze utente specifiche, i servizi applicativi non potrebbero funzionare senza i programmi di supporto, cioè quei programmi che consentono di preparare l'ambiente e forniscono le funzionalità di servizio; chiamiamo questi programmi **componenti di sistema** (non solo il sistema operativo). In generale, i componenti di sistema tendono ad essere eseguiti molte volte e molto spesso, perciò devono essere ottimizzati; in particolare, nei sistemi distribuiti si tende ad avere sempre più componenti di sistema che si devono coordinare tra di loro in modo dinamico e veloce.

1.2. Programmazione a microservizi

L'esigenza di avere componenti di sistema è stata spesso espressa in modo monolitico: un programma è sempre lo stesso e contiene tutte le funzioni necessarie; se attiviamo più volte lo stesso programma con più incarnazioni o processi, possiamo avere molti possibili servizi applicativi contemporaneamente, però più processi comportano la ripetizione delle parti di supporto, in modo non scalabile: si deve pensare a strutture ripetute per ogni processo, sia per la parte di supporto, sia per la parte applicativa. L'esigenza di avere componenti che si devono coordinare tra loro in modo dinamico, veloce e ottimizzato all'uso ha portato a definire i microservizi, programmi piccoli, efficienti e capaci di interagire in modo dinamico che si possano comporre al bisogno e con costi limitati. I microservizi si collegano alla programmazione di sistema in quanto contengono sia la parte applicativa che quella di sistema. Questa tendenza che impacca insieme l'applicazione con tutto quello che è necessario per supportarla in un unico piccolo servizio è una direzione fortemente propagata del filone detto DEVOPS, ossia tutta la tecnologia che mette attenzione sulla parte di supporto e lo considera integrante e da integrare nella definizione dell'applicazione. In generale, ogni programma che si debba mettere in esecuzione ha bisogno di risorse di sistema di basso e alto livello; anche nei sistemi concentrati quindi la consapevolezza delle risorse è un obiettivo da perseguire: è necessario usare le risorse dell'architettura in modo ottimale, senza sprechi e con un buon controllo delle situazioni di errore, evitando l'utilizzo di risorse complesse, pesanti e difficili da implementare e portare tra architetture.

1.3. Architettura delle soluzioni

In generale, nell'espressione dei programmi ci sono due paradigmi ispiratori:

- la **programmazione imperativa**: descrive i linguaggi tradizionali, anche ad oggetti, con la specifica di algoritmi precisi di soluzione; le istruzioni specificano completamente l'algoritmo da risolvere senza libertà di cercare soluzioni non specificate. L'algoritmo viene specificato come una sequenza precisa di passi, che vengono eseguiti uno dopo l'altro, e che agisce sui dati, intesi come contenitori in memoria delle informazioni; in genere è poi il sistema operativo che gestisce la possibilità di concorrenza. I linguaggi, come ad esempio C, C++, C#, Java, utilizzano uno stato, inteso come insieme di variabili, che è manipolato dagli algoritmi;
- la **programmazione dichiarativa**: fa riferimento a modi di esprimere la computazione in modo meno deterministico. Questo paradigma porta ad esprimere soluzioni che devono soddisfare una serie di requisiti specificati dall'utilizzatore, senza arrivare ad una specifica completa di algoritmo, che spesso può esplorare soluzioni molteplici e viene guidato da non determinismo e da modalità ad elevato innestamento, funzionale o logico, garantito da un motore alla base del supporto.

Per lo sviluppo di un'applicazione inizialmente si decide il design dell'algoritmo di soluzione e successivamente lo si codifica in uno o più linguaggi di programmazione con l'obiettivo di arrivare alla

sua esecuzione. Per arrivare all'esecuzione, è necessario stabilire quali sono le risorse hardware e fisiche su cui poter eseguire e attuare una corrispondenza tra la parte logica e la parte concreta di architettura; la scelta dell'architettura, con la sua specifica configurazione, è la scelta di deployment, ovvero la fase in cui si decide come allocare le risorse logiche alle risorse fisiche disponibili.
L'architettura di supporto si divide in due fasi:

- **fase statica**: si inizia con l'analisi, il progetto dell'algoritmo e la sua codifica in uno più linguaggi di programmazione;
- **fase dinamica**: si decide l'architettura concreta e si carica il programma per quella configurazione. A questo punto si inizia la reale esecuzione del programma, da cui ottenere dati e la miglior performance. Questa fase la si vuole molto efficiente, con la possibilità di usare al meglio le risorse e non sprecarle.

Nel caso di programmi di sistema è necessario saper mettere in relazione i rapporti tra i diversi livelli, e il modo in cui richiedono risorse, fino all'interazione con le risorse locali. Alcuni aspetti base da considerare durante il progetto di sistema sono la semplicità, l'efficienza e l'orientamento ai requisiti; un'esecuzione efficiente richiede anche abilità nella parte di progetto in modo da avere componenti di cui si possano fare deployment efficienti in ogni ambiente. Nel caso di programmi di sistema che devono essere parte del supporto e magari eseguiti milioni di volte, il progetto deve mirare a:

- semplicità delle strutture dati e degli algoritmi;
- minimizzazione del costo della configurazione;
- minimizzazione dell'impegno sulle risorse;
- minimizzazione dell'overhead;
- capacità di interazione con l'utente;
- prevenzione errori;
- capacità di controllo dell'esecuzione.

1.4. Semplicità degli algoritmi e del codice

Gli algoritmi devono essere semplici:

- occorre eliminare controlli ridondanti per ridurre i costi in eccesso, come ad esempio le funzioni sulle stringhe in C che non controllano il formato;
- occorre avere poco innestamento di procedure e funzioni non necessarie in modo da diminuire l'utilizzo dello stack;
- vanno eliminate le ripetizioni, ad esempio usando variabili per tenere lo stato di un'esecuzione invece di ripetere la computazione più volte.

Nel caso di processi e programmi che interagiscono con l'utente, occorre:

- controllare i parametri di invocazione e procedere solo se corretti;
- verificare gli input dell'utente e non procedere in caso di errore per evitare azioni inutili e costose;
- consumare gli stream di input e le risorse in modo corretto;
- interagire usando un limitato numero di risorse di sistema;
- progettare bene i processi.

2. Generalità, obiettivi e modelli di base

2.1. Introduzione ai sistemi distribuiti

I sistemi distribuiti sono sistemi residenti in località diverse che usano la comunicazione e la cooperazione per ottenere risultati coordinati. Sono sistemi più complessi di quelli concentrati ma la loro esigenza è molto sentita in quanto permettono l'accesso a risorse remote da ovunque e permettono la loro condivisione come se fossero locali. Ciò che si desidera ottenere in un sistema distribuito è la trasparenza dell'allocazione, la dinamicità del sistema e la qualità dei servizi; i problemi che si incontrano sono causati dalla complessità dei sistemi: occorre gestire la concorrenza, in quanto moltissimi processi possono essere eseguiti contemporaneamente, non esiste un tempo globale per sincronizzare il sistema e possono esserci fallimenti e crash sui singoli nodi della rete. Lo sviluppo di un'applicazione distribuita prevede l'analisi, lo sviluppo sulla base di un algoritmo, la sua codifica in linguaggi di programmazione e l'esecuzione finale; le due fasi principali sono:

- **mapping**: è la fase in cui si stabilisce la configurazione per l'architettura e si stabilisce come allocare le risorse logiche sulle risorse fisiche e le località;
- **binding**: è la fase in cui si stabilisce come ogni entità dell'applicazione si lega alle risorse del sistema; può essere:
  - **statico**: si stabilisce a priori come legare le entità logiche alle risorse del sistema;
  - **dinamico**: dato che non si può prevedere come allocare le risorse prima dell'esecuzione, lo si fa dinamicamente durante l'esecuzione e si utilizzano dei sistemi di nomi per poter riferire in modo efficiente i processi con cui si vuole comunicare.

Per un'applicazione distribuita sono necessari i processi, delle azioni specifiche da eseguire sui dati locali e dei protocolli per ottenere le azioni coordinate; rispetto ai sistemi concentrati si aggiungono i protocolli, ovvero un complesso di regole e procedure a cui ci si deve attenere in determinate attività per l'esecuzione corretta. UNIX standardizza i processi e mette a disposizione API per:

- interagire con i file (open, read, write, close);
- progettare filtri con una corretta gestione delle risorse;
- poter avere la concorrenza dei processi (fork, exec);
- poter gestire la comunicazione tra processi.

2.2. Processi

I processi possono essere pesanti o leggeri; nella loro implementazione, i processi sono un'aggregazione di parecchi componenti e hanno uno spazio di indirizzamento e uno spazio di esecuzione. I processi pesanti richiedono molte risorse e il loro cambio di contesto è un'operazione molto pesante con molto overhead (ovvero richiedono molte più risorse del minimo necessario), soprattutto per la parte di sistema. I processi leggeri permettono di ovviare a questi problemi: condividono tra di loro la visibilità di un ambiente contenitore e sono caratterizzati da uno stato limitato e da un overhead limitato; il contenitore unico è un processo pesante che fornisce la visibilità comune a tutti i thread. Java supporta i processi leggeri: i thread creati sono tutti mappati all'interno di un processo pesante, la JVM.

2.3. Modello Client/Server

Il modello client/server è un modello di coordinamento tra due entità che cooperano per una comunicazione e un servizio; l'implementazione è molto varia e può comportare scelte e politiche molto diverse ma è sempre molti a 1, ovvero c'è un server e tanti client. In questo modello i client chiedono, ovvero richiedono un servizio, e il server risponde. Il modello può essere:

- **sincrono** se è prevista una risposta (rispettivamente asincrono se la risposta non è prevista);
- **bloccante** se si attende la risposta del pari (rispettivamente non bloccante se non la si attende);

- **asimmetrico** se il client conosce a priori il server ma il server non conosce a priori il client (rispettivamente simmetrico se entrambi si conoscono a priori);
- **dinamico** se il binding tra client e server è dinamico, ovvero il server che risponde alle richieste può cambiare tra le diverse invocazioni.

A default il modello è sincrono, bloccante, asimmetrico e dinamico. Progettare il server è più complesso del client in quanto deve essere sempre pronto a eventuali richieste; il server deve quindi essere un demone, ovvero un processo sempre attivo in un ciclo infinito. Il server, oltre alla comunicazione, deve realizzare il servizio con azioni locali, come ad esempio accedere alle risorse del sistema, considerando molteplici client e anche problemi di integrità dei dati, accessi concorrenti, autenticazione utenti, autorizzazione all'accesso e privacy delle informazioni.
Il client tipicamente lavora in modo sincrono e con interazione bloccante: c'è sempre una risposta e il client la aspetta. Per ovviare ad attese troppo lunghe (o magari infinite) per una risposta da parte del server si usa un time-out al cui scadere scatta un'eccezione. In questo caso si può pensare di rifare una trasmissione o di cambiare server, infatti un server potrebbe essere lento e congestionato a causa di richieste precedenti.

2.4. Modelli asincroni – pull e push

Il modello **pull**, anche detto a polling, è un modello in cui un client chiede ripetutamente lo stesso servizio ad un server impostando un time-out brevissimo per la risposta, fino a quando non riceve una risposta in tempi accettabili. In questo modello, sincrono non bloccante, il server è più semplice da progettare e il client decide quando ripetere la richiesta, quanto spesso e quante volte: il client ha sempre l'iniziativa. Questo modello è utile nel caso in cui si hanno server spesso congestionati e lenti, in modo tale da non dover far attendere troppo a lungo i client.
Nel modello di interazione **push** invece il client fa la richiesta una volta sola, si sblocca e può fare altro; il server arriva a eseguire il servizio richiesto e ha la responsabilità della consegna del risultato al client. Il modello push fa diventare il server cliente di ogni suo client, scaricando il client ma caricando di ulteriori compiti il server stesso.

2.5. Modello a delegazione

In caso di interazione sincrona non bloccante, si possono delegare le funzionalità di ricezione del risultato ad un'entità che opera al posto del responsabile e lo libera di un compito: questa entità è detta proxy. In questo modello un client lascia il proxy ad aspettare la risposta di una richiesta fatta ad un server lento e il proxy lavora in modo pull o push per fornire la risposta al client.

2.6. Modello ad eventi

Questo modello è diverso dal modello C/S per molti aspetti; innanzitutto, questo modello è molti a molti e non più molti a 1: sono presenti molti client, molti server e un gestore centrale che gestisce la comunicazione tra i vari client e i vari server. Questo modello è anche detto modello pub/sub: i server, detti publisher, pubblicano i loro servizi e i client, detti subscriber, si iscrivono al gestore; i client sono disaccoppiati ed è il gestore degli eventi che segnala l'occorrenza degli eventi e invia i relativi messaggi agli interessati. Questo modello, a differenza del modello C/S, presenta uno scarso accoppiamento tra client e server e non impone la compresenza delle entità interagenti.

2.7. Modello ad agenti multipli

In questo modello i servizi sono forniti dal coordinamento di più server, detti agenti, che forniscono un servizio globale unico. Gli agenti forniscono il servizio coordinato e possono partizionare le capacità di servizio e replicare le funzionalità di servizio, in modo trasparente al client.

2.8. Stato

Nell'interazione C/S, un aspetto fondamentale è lo stato dell'interazione, ovvero occorre tenere traccia della comunicazione e delle azioni svolte precedentemente; in caso di stato, infatti, l'interazione è facilitata essendo riconosciuta e gestita. Il rapporto C/S può quindi essere:

- **stateless**: non si tiene traccia dello stato. Ogni messaggio è completamente indipendente dagli altri;
- **stateful**: si mantiene lo stato dell'interazione tra chi interagisce: un messaggio e l'operazione conseguente può dipendere da ciò che è avvenuto precedentemente. In genere, il client che fa una richiesta tende a tenere traccia dello stato in caso di azioni ripetute.

Lo stato dell'interazione consente di avere un accordo predefinito tra le entità interagenti e di avere una gestione condivisa che facilita la comunicazione; il client può quindi semplificare il protocollo, assumendo che il server lo faciliti mantenendo uno stato. Lo stato dell'interazione è solitamente memorizzato nel server:

- un server stateless è più leggero e più affidabile in presenza di malfunzionamenti ma lo stato deve essere mantenuto da ogni client;
- un server stateful garantisce efficienza in quanto le dimensioni dei messaggi sono più contenute e si ottiene una migliore velocità di risposta del server.

Un'interazione stateless è sensata e possibile solo se il protocollo è progettato per operazioni idempotenti, ovvero operazioni che anche dopo un infinito numero di esecuzioni, daranno sempre lo stesso risultato.

2.9. Tipi di server

Il server è tipicamente una sola attività e un solo processo ma la concorrenza può migliorare le sue prestazioni. Un server può essere:

- **sequenziale** (anche detto iterativo): le richieste di servizio vengono processate una alla volta, mettendo in coda di attesa le altre. Può esserci un possibile basso utilizzo delle risorse, in quanto c'è sovrapposizione tra elaborazione ed I/O. Per limitare l'overhead si può limitare la lunghezza della coda e rifiutare le richieste a coda piena;
- **concorrente**: può gestire molte richieste di servizio insieme (in modo concorrente ma non per forza in parallelo), cioè accettare una richiesta prima del termine di quella in corso di servizio. Si ottengono migliori prestazioni grazie alla sovrapposizione di elaborazione ed I/O ma una maggiore complessità progettuale. Un server concorrente può essere sia mono processo, in cui un unico processo server si divide tra il gestire la coda delle richieste e le operazioni vere e proprie, e sia multi-processo, in cui un processo server si occupa della coda delle richieste e genera un processo figlio per ciascun servizio da eseguire.

Nel progetto di una iterazione si sceglie in base alle caratteristiche tecnologiche, come ad esempio il sistema operativo di supporto. In ambiente UNIX è possibile generare processi pesanti in modo facile e semplificato e si possono utilizzare server multi-processo per avere la concorrenza; in ambiente Java per gestire server multi-processo è possibile generare thread.

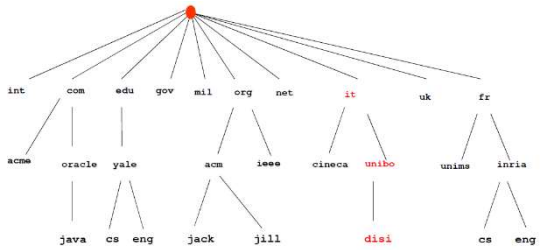
2.10. Sistemi di nomi

Per accedere ai servizi è necessario poterli identificare e trovare. Questa funzione è svolta dai sistemi di nomi, ossia server capaci di fornire servizi di gestione e mantenimento dei nomi. Un client che vuole richiedere un servizio deve poter riferire il server: questo è reso possibile tramite il nome che usa il client per riferire il server, ovvero il nome logico, un nome trasparente rispetto alla locazione del server che permette di avere un più alto livello di astrazione. Il binding dei nomi può essere statico o dinamico: nel caso statico i riferimenti sono risolti prima dell'esecuzione, nel caso dinamico i riferimenti sono risolti solo al momento del bisogno e durante l'esecuzione. Nel caso distribuito le risorse sono dinamiche e non sono prevedibili staticamente, quindi il binding è dinamico durante l'esecuzione. Un sistema di nomi

deve rendere possibile inserire nuove entità durante l'esecuzione, oltre a quelle già esistenti, e consentire di ritrovare rapidamente il nome delle entità; il sistema di nomi va quindi organizzato con un'architettura che permetta fattibilità ed efficienza attraverso molteplici gestori, che possono essere:

- **gestori partizionati**: ciascuno è responsabile di una sola parte, cioè una partizione, dei riferimenti di località: sono presenti più gestori per rendere più efficiente il servizio;
- **gestori replicati**: ciascuno è responsabile, insieme ad altri, di una parte dei riferimenti di coordinamento, per far fronte a guasti di uno dei gestori dell'insieme.

2.11. DNS – *Domain Name System*



DNS è un servizio di nomi basato su un insieme di gestori coordinati che si organizzano per rispondere a query che chiedono l'indirizzo IP, ovvero l'indirizzo fisico, corrispondente ad un nome di dominio. I gestori si occupano di gestire le tabelle formate da {nome logico, nome fisico} attuando una corrispondenza tra questi. In questo modo un client che deve raggiungere il server non deve sapere il suo indirizzo IP e non deve sapere dove si trova: gli basta il suo indirizzo logico e tramite un DNS è possibile risalire all'indirizzo IP. DNS introduce i nomi logici in una gerarchia (un albero), intesa come gerarchia di domini logici e centri di servizio: la corrispondenza tra nomi logici e indirizzi fisici avviene tramite un servizio di nomi che risponde dinamicamente alle richieste.
Una richiesta viene smistata da un agente specifico, detto **name resolver**. Ogni dominio ha un name resolver, sistemi di nomi locali e fa un ampio uso di una memoria cache per memorizzare le associazioni pregresse, quindi quando un client chiede un mapping al name resolver, quest'ultimo fornisce la risposta o perché conosce già la corrispondenza tramite la cache o la trova attraverso una richiesta C/S ad un name server; grazie all'organizzazione ad albero è possibile muoversi tra i vari DNS con le richieste fino a raggiungere il dominio di autorità. I client e i name resolver fanno richieste usando il protocollo UDP sulla porta 53 e in caso di messaggi troppo lunghi si usa TCP. Il protocollo tra server DNS ha due tipi di query:

- **iterativa**: prevede una sequenza di richieste request/reply; si fornisce al richiedente o la risposta o un riferimento al miglior DNS server che possa averla;
- **ricorsiva**: prevede una catena di server request/reply. Comporta che al richiedente si fornisca una risposta, anche chiedendo ad altri, o si segnali un errore (ad esempio di dominio non esistente).

3. Programmazione di rete e modello C/S con socket in Java

3.1. Introduzione

Grazie alle socket, che rappresentano l'end-point locale di un canale di comunicazione bidirezionale, è possibile far comunicare tra loro macchine distinte, diverse e fortemente eterogenee. In Java è possibile usare le socket mediante classi specifiche del package di networking java.net. La comunicazione può essere senza connessione, usando UDP e le socket datagram, e con connessione, usando TCP e le socket stream. Per identificare univocamente un processo su una macchina con il quale si vuole comunicare si usa la coppia {indirizzo IP, porta}, in cui l'indirizzo IP è formato da 32 bit e la porta è un numero intero di 16 bit: i messaggi sono consegnati su una specifica porta di una specifica macchina, non direttamente ad un processo: sono le socket che legano il processo ad un nome globale per ricevere o spedire dei messaggi; con questo doppio sistema di nomi è possibile identificare un processo senza conoscere il suo PID locale. In Java i server possono essere sia sequenziali con/senza connessione, sia paralleli, con un processo master che genera un thread per ogni servizio da gestire.

3.2. Socket datagram

Le socket datagram permettono a due processi di scambiarsi messaggi senza stabilire una connessione tra di loro. Esiste un solo tipo di socket datagram sia per i client che per i server, definita nel package java.net.DatagramSocket: **public final class DatagramSocket extends Object**. Un possibile costrutture è **DatagramSocket(InetAddress localAddress, int localPort) throws SocketException**, in cui si crea una socket UDP e si fa un binding locale a una specifica porta e a uno specifico indirizzo IP. I due metodi che permettono di inviare e ricevere datagrammi sono rispettivamente **void send(DatagramPacket p)** e **void receive(DatagramPacket p)**: la send implica la consegna ad un livello di kernel locale che si occupa solamente dell'invio, asincrono con la ricezione; la receive, che assume che la vera ricezione sia delegata al kernel, richiede un'attesa del ricevente fino all'arrivo locale dell'informazione: basta ricevere un datagramma per sbloccare una receive. La classe DatagramPacket serve per preparare e usare i datagrammi, che specificano cosa comunicare e con chi; il suo costruttore è **DatagramPacket(byte[ ] buf, int offset, int length, InetAddress address, int port)**, in cui, in ordine, si ha un array con i dati da inviare/ricevere, l'indirizzo di inizio, la lunghezza dei dati e l'indirizzo IP e la porta da cui leggere/scrivere.

3.3. Socket multicast

La comunicazione tramite datagrammi permette anche la comunicazione multicast, ovvero l'invio di uno stesso messaggio a una serie di destinatari che sono registrati su un indirizzo di gruppo (di ricezione). Si crea una socket multicast con il costruttore **MulticastSocket(int multicastPort)** e si può entrare ed uscire dal gruppo con i metodi **joinGroup(InetAddress address)** e **leaveGroup(InetAddress address)**.

3.4. Socket stream

Le socket stream permettono di avere un canale di comunicazione virtuale creato prima della comunicazione: la comunicazione è bidirezionale, affidabile e con i dati consegnati una volta sola (semantica at-most-once di TCP). La connessione tra i processi client e server è definita da una quadrupla univoca {indirizzo IP 1, porta 1, indirizzo IP 2, porta 2} e dal protocollo TCP. A differenza delle socket datagram, per le socket stream in Java sono presenti due tipi di socket distinte, una per i client e una per i server: java.net.Socket e java.net.ServerSocket.
Per i client, la classe Socket permette di creare una socket connessa: i costruttori della classe creano la socket, la legano a una porta locale e la connettono alla porta di una macchina remota su cui risiede il

server. Esistono più costruttori, come ad esempio **public Socket(InetAddress remoteHost, int remotePort) throws IOException** per creare una socket stream e collegarla alla porta specificata e all'indirizzo IP specificato, riconoscendo in automatico l'indirizzo IP della macchina locale e usando una porta a caso, e **public Socket(InetAddress remoteHost, int remotePort, InetAddress localHost, int localPort) throws IOException** per creare una socket client specificando tutti i parametri.
Per i server, le ServerSocket permettono unicamente di accettare richieste di connessione provenienti dai vari client. Per creare una socket dal lato server si possono usare i due costruttori **public ServerSocket(int localPort) throws IOException** e **public ServerSocket(int localPort, int count)**, che si utilizza nel caso si voglia specificare la dimensione della coda delle richieste, che di default è 5. Il server si deve mettere in attesa di nuove richieste di connessione usando la primitiva **public Socket accept() throws IOException**, che blocca il server fino all'arrivo di una richiesta di connessione e, quando avviene, restituisce un oggetto della classe Socket su cui avviene lo scambio di byte vero e proprio tra client e server. La chiamata di accept è sospensiva in attesa di richieste di connessione: se non ci sono ulteriori richieste, il server si blocca in attesa e se c'è almeno una richiesta si sblocca e si crea la connessione. In caso di server parallelo, all'accettazione il server può generare un thread responsabile del servizio: questo eredita la connessione e la chiude al termine della comunicazione; il server principale può tornare immediatamente ad aspettare nuove richieste e servire nuove operazioni.
Per leggere e scrivere sulle socket stream si usano i metodi **public InputStream getInputStream()** e **public OutputStream getOutputStream()**, che restituiscono un oggetto stream che incapsula il canale di comunicazione. Attraverso gli stream generici di byte si possono soltanto spedire e ricevere byte, senza nessuna formattazione dei messaggi: si possono usare gli stream DataInputStream e DataOutputStream per avere funzionalità di più alto livello. Alcuni metodi di queste classi sono **void writeUTF(String str)**, **String readUTF()**, **void writeInt(int v), int readInt()**. Si noti che si usa lo standard UTF, *Unicode Transformation Standard*, per consentire le diverse internazionalizzazioni necessarie e poter gestire in modo uniforme le stringhe, infatti il supporto JVM memorizza le stringhe con un formato interno UTF-16.

3.5. Chiusura e opzioni per le socket

Una volta create e usate le socket, quando non più necessarie è opportuno chiuderle in modo da evitare di usare risorse inutilmente: a questo scopo si usa il metodo **public synchronized void close() throws SocketException** che permette di chiudere l'oggetto socket e disconnettere il client dal server. Quando si chiude una socket, si eliminano immediatamente i dati nella memoria IN e si mantengono per un certo periodo di tempo i dati nella memoria OUT, in modo da poter finire l'invio dei dati al pari; quando si chiude una socket il pari si accorge di questo tramite eccezioni o eventi che gli vengono notificati in caso di operazioni di lettura e scrittura sulla socket chiusa. Per non chiudere completamente la socket esistono anche le primitive **shutdownInput()** e **shutdownOutput()**, che permettono, rispettivamente, di chiudere soltanto il canale IN e soltanto il canale OUT. Quella più usata è la shutdownOutput(), in quanto ogni partecipante della comunicazione dipende dall'altro per la lettura.
Per quanto riguarda le socket datagram esistono diverse opzioni per cambiare il loro comportamento a default. Ad esempio, la ricezione da socket tramite la receive è sincrona bloccante: usando il metodo **SetSoTimeout(int timeout)** si può impostare un time-out dopo il quale l'operazione termina; tramite **SetSendBufferSize(int size)** e **SetReceiveBufferSize(int size)** si può cambiare la dimensione del buffer di invio e ricezione del driver.
Anche le socket stream hanno diverse opzioni, descritte nell'interfaccia SocketOptions: tramite il metodo **SetSoLinger(boolean on, int linger)** è possibile impostare dopo quanto tempo, detto intervallo di linger, vengono scartati i pacchetti in output sulla socket; tramite il metodo **SetTcpNoDelay(boolean**

on) si può inviare direttamente un pacchetto senza bufferizzarlo e tramite **SetKeepAlive(boolean on)** si abilitano e disabilitano le opzioni di keepalive.

4. Programmazione di rete e modello C/S con socket in C

4.1. Introduzione

Nel caso locale, per la comunicazione e la sincronizzazione si usano i segnali, con cui un processo invia un evento ad un altro processo, e i file, usabili solo tra processi che condividono il file system sullo stesso nodo. Nel caso specifico di processi coresidenti sullo stesso nodo, ci sono tre possibili soluzioni:

- le pipe, usabili solo tra processi con un parente in comune;
- le pipe con nome, usabili per i processi su una stessa macchina;
- la memoria condivisa, usabile su una stessa macchina.

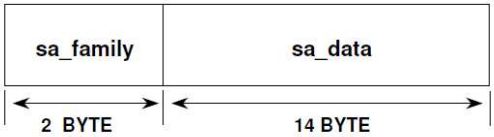
Per la comunicazione e la sincronizzazione remota in C si usano le socket, che permettono di comunicare in modo flessibile, differenziato ed efficiente. In UNIX ogni processo mantiene una tabella dei file aperti, in cui ogni sessione aperta sui file viene mantenuta attraverso uno specifico file descriptor e il paradigma d'uso è open, read/write, close; le socket sono conformi a questo paradigma e sono ben integrate con i processi e i file: hanno un'interfaccia omogenea con quella tradizionale di UNIX e questo permette di invocare i servizi in modo trasparente. Le proprietà delle socket sono:

- **eterogeneità**: è possibile comunicare fra processi anche su architetture diverse;
- **trasparenza**: la comunicazione fra processi è indipendente dalla localizzazione fisica;
- **efficienza**: l'applicabilità delle socket è limitata dalla sola performance;
- **compatibilità**: i naive process (filtri) possono lavorare in ambiti distribuiti senza subire alcuna modifica;
- **completezza**: si possono usare protocolli di comunicazione diversi e differenziati.

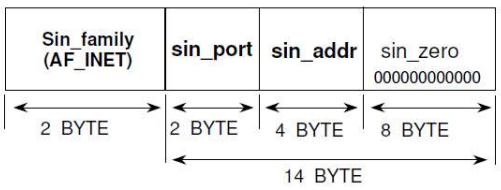
4.2. Domini di comunicazione, nomi e primitive preliminari

Esistono più domini di comunicazione, come ad esempio UNIX e Internet. La prima scelta che si fa è il protocollo di comunicazione PF, *Protocol Family*, tra PF_UNIX per comunicare localmente tramite le pipe e PF_INET per comunicare mediante i protocolli di Internet. Successivamente occorre specificare l'AF, *Address Family*, ovvero la famiglia di indirizzi che si sta utilizzando: AF_UNIX permette di usare indirizzi per la comunicazione locale mentre AF_INET permette di usare indirizzi per Internet. Le socket, per poter essere utilizzate, richiedono un binding tra il loro nome logico, che è un nome locale, e il loro nome fisico, ovvero la porta su cui sono accessibili, che è il nome globale. Una half-association è definita come una coppia di nomi formata da un nome logico e un nome fisico, che nel caso del dominio Internet si ottiene con la famiglia dell'indirizzo, l'indirizzo Internet e il numero della porta.
Per rappresentare i nomi delle socket, ovvero gli indirizzi, si usano due tipi di strutture:

- **sockaddr**: serve per rappresentare un indirizzo generico. La sua dimensione è di 16 byte ed è formata nel seguente modo: (u_short sa_family, char sa_data[14]);



- **sockaddr_in**: serve per un indirizzo specifico della famiglia AF_INET. Anche questa struttura è di 16 byte ed è formata nel seguente modo: (u_short sin_family, u_short sin_port, struct in_addr sin_addr); alla fine della struttura sono anche presenti 8 byte, non utilizzati, detti sin_zero.



Un client conosce il nome logico Internet, ovvero una stringa, di un server remoto ma non conosce il nome fisico corrispondente. Per poter ottenere il nome fisico partendo da un nome logico si può usare

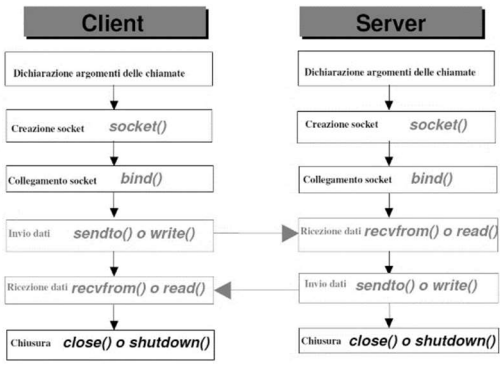
la primitiva **struct hostent* gethostbyname(char* name)** che restituisce un puntatore alla struttura hostent oppure NULL in caso di fallimento. La ricerca avviene localmente e successivamente viene integrata anche da sistemi di nomi (ad esempio DNS). Tramite la struttura hostent si possono ricavare tutte le informazioni riguardo un nodo di cui si ha il nome logico; quelle più rilevanti sono il nome fisico primario (il primo nella lista) e la sua lunghezza, che in Internet è fissa. In modo simile, per consentire ad un utente di usare dei nomi logici di servizio senza ricordare la porta, la funzione **struct servent* getservbyname(char* name, proto* protocol)** restituisce il numero di porta relativo ad un servizio.
Per poter lavorare sulle socket bisogna utilizzare due primitive preliminari, entrambe essenziali:

- **socket(dominio, tipo, protocollo)**: permette di creare la socket; restituisce il file descriptor della socket creata;
- **bind(sd, nome, lungnome)**: permette di agganciare la socket ad una porta fisica; restituisce un valore positivo se è andata a buon fine.

4.3. Presentazione dei dati

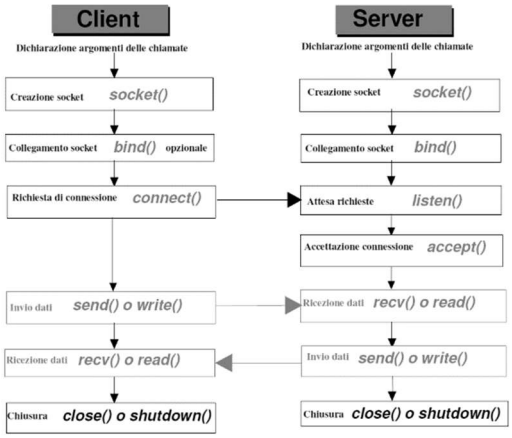
Gli interi sono composti da più byte e possono essere rappresentati in memoria secondo due modalità diverse di ordinamento: **little-endian**, in cui il byte di ordine più basso si trova all'inizio e **big-endian**, in cui il byte di ordine più alto si trova all'inizio. L'ordinamento seguito dai vari host, detto HBO, *Host Byte Order*, può variare, ad esempio Intel adopera l'ordinamento little-endian e Solaris big-endian. Internet invece adopera un ordinamento detto NBO, *Network Byte Order*, che è big-endian. Per risolvere questi problemi di eterogeneità si adoperano le funzioni accessorie **htons()** e **htonl()** (*Host to Network Short/Long*) per convertire da HBO a NBO e le funzioni **ntohs()** e **ntohl()** (*Network to Host Short/Long*) per convertire da NBO a HBO.

4.4. Socket datagram



Le socket datagram sono dei veri end-point di comunicazione e permettono di formare half-association, relative ad un solo processo, ma usabili per comunicare con chiunque del dominio. Per scambiare i datagrammi occorre, da parte del processo client, dichiarare le variabili di riferimento a una socket, conoscere l'indirizzo Internet del nodo remoto e conoscere la porta del servizio da usare; per il processo server invece occorre dichiarare delle variabili di riferimento a una socket, conoscere la porta per il servizio da offrire e occorre poter ricevere su qualunque indirizzo IP locale, utile per i server multi-porta (con più connessioni). Per la comunicazione, oltre alle funzioni read() e write(), usabili per mantenere l'omogeneità con i file in UNIX, sono disponibili anche le funzioni **sendto()** e **recvfrom()**, le quali restituiscono il numero dei byte inviati e ricevuti.
La firma di queste due funzioni è (sd, message, length, flags, to/from, tolength/fromlength), che in ordine sono: il socket descriptor su cui si scrive/legge, un puntatore al messaggio da inviare/ricevere e la sua lunghezza, dei flag per impostazioni varie, un puntatore alla socket partner e alla sua lunghezza. La lunghezza massima dei messaggi che si scambiano è di 9 kB o 16 kB e sono usati i protocolli UDP e IP, intrinsecamente non affidabili, quindi non è detto che tutti i datagrammi inviati arrivino effettivamente al ricevente. Per evitare di perdere parti di un messaggio si cerca quindi di riceverli in un'area di memoria grande e per massimizzare l'affidabilità è buona norma diminuire il numero di datagrammi, se possibile, e usare dei messaggi di conferma.

4.5. Socket stream



Le socket stream prevedono una risorsa che rappresenta la connessione virtuale tra le entità interagenti. È presente un'entità attiva, ovvero il client, che richiede il servizio e un'entità passiva, il server, che accetta il servizio e risponde. È presente una prima fase asimmetrica, ovvero quella di accettazione del client, e poi una fase simmetrica in cui client e server comunicano. La connessione, una volta stabilita, permane fino alla chiusura di una delle due half-association, ossia fino alla decisione di una delle due entità di chiudere la connessione. I processi naive possono sfruttare le socket stream e lavorare in modo trasparente in remoto leggendo da input e scrivendo su output. Nei client la bind() è opzionale in quanto non hanno necessità di essere visibili dall'esterno, a differenza dei server, quindi la bind è implicita nella connect(). La primitiva **connect()** è una primitiva di comunicazione sincrona e termina quando la richiesta è accodata o in caso di errore. Al termine della connect() la connessione è creata. In caso di errore, ovvero quando restituisce un risultato negativo, si può scoprire la motivazione del problema nel valore della variabile *errno*:

- ECOMM: errore di comunicazione nell'invio;
- ECONNABORTED: la connessione è stata interrotta;
- ECONNREFUSED: non è possibile connettersi;
- ETIMEDOUT: il tentativo di connessione è in time-out. La coda di ascolto del server è piena o non è stata creata, quindi non è stata depositata la richiesta.

La connect() ha successo e restituisce il controllo quando ha depositato la richiesta nella coda del server. Il server deve creare una coda per possibili richieste di servizio e lo fa mediante la primitiva **listen()**, che ha come argomenti il socket descriptor e il numero di richieste massime gestibili. La primitiva listen() è una primitiva locale, istantanea e senza attesa; fallisce solo se attuata su socket non adatte. Al termine della listen(), la coda è disponibile per accettare richieste di connessione nel numero specificato. Le richieste che vanno oltre alla coda sono semplicemente scartate. Le richieste che si trovano in coda vanno gestite con la primitiva **accept()**, una primitiva locale, con attesa e correlata alla comunicazione con il client: se ha successo produce la vera connessione, se fallisce, in caso di socket non adatte, non consente di proseguire. Per ricevere e inviare byte si possono usare le primitive **recv()** e **send()**, che restituiscono il numero di byte effettivamente ricevuti o inviati. La firma di queste due funzioni è: (sd, message, length, flags) in cui, in ordine, rappresentano il socket descriptor su cui si comunica, un puntatore all'area che contiene il messaggio da inviare o ricevere, la lunghezza del messaggio e le opzioni di comunicazione. Per sfruttare al meglio la rete è opportuno che ogni dato inviato sia ricevuto, evitando di dover scartare dati che sono arrivati ad un end-point; nelle socket stream i messaggi non sono comunicati ogni volta che si usa una primitiva ma i dati sono bufferizzati dal protocollo TCP, quindi vengono raggruppati e inviati poi alla prima comunicazione decisa dal driver TCP: per ovviare a questo problema si possono mandare messaggi di lunghezza pari al buffer o usare l'opzione watermark. Ogni primitiva di ricezione restituisce i dati del driver locali e TCP non implementa marcatori di fine messaggio: occorre quindi mandare messaggi di lunghezza fissa; per mandare messaggi a lunghezza variabile, si alternano un messaggio a lunghezza fissa e uno a lunghezza variabile: il primo deve contenere la lunghezza del secondo.

4.6. Chiusura delle socket

Su ogni flusso viaggiano dati in stream fino alla fine del file del pari di autorità che, dopo aver operato completamente, segnala con una chiusura che non ci sono più dati e si aspetta che i dati siano consumati:

la chiusura causa un'indicazione di fine file, ovvero EOF, all'altro. Ogni end-point deve leggere tutto l'input fino alla fine del flusso e deve inviare una fine del flusso quando vuole terminare i dati in uscita. Per evitare di utilizzare risorse non necessarie, occorre chiudere le socket non più utilizzate con la primitiva **close()**, che è locale, passante ed è istantanea per il processo che la invoca. Questa primitiva decrementa il contatore dei processi referenti al socket descriptor, quindi il chiamante non può più usare quel descrittore. Ogni socket stream è associata ad un buffer di memoria per mantenere i dati in uscita e i dati in ingresso; alla chiusura, ogni messaggio nel buffer associato alla socket in uscita sulla connessione deve essere spedito, mentre ogni dato in ingresso ancora non ricevuto viene buttato via; solo dopo si può deallocare la memoria del buffer di trasporto. Dopo aver effettuato una close, il pari connesso alla socket chiusa se legge, ottiene EOF, mentre se scrive ottiene un segnale di connessione non più esistente. In alternativa alla close(), per una migliore gestione delle azioni tra i pari, si può usare la primitiva **shutdown()**, con la quale si può chiudere soltanto un canale (o input o output). Ognuno dei due pari gestisce il proprio verso di uscita e lo controlla: se decide di finire, usa una shutdown dell'output e segnala di non voler più trasmettere; il pari legge fino a EOF e poi sa di non doversi più occupare dell'input. Il contratto TCP prescrive che i dati in input vengano buttati via subito; i dati in output vengono invece consegnati al pari che è tenuto ad aspettarli e riceverli fino a che non riceve EOF; la consegna è garantita e potrebbe durare anche molto tempo, quindi questo potrebbe impedire l'aggancio alla stessa porta da parte di un nuovo processo locale.

4.7. Opzioni per le primitive

Grazie alle funzioni **getsockopt()** e **setsockopt()** è possibile configurare le socket e variare la loro modalità di utilizzo. Alcune opzioni possibili sono:

- **SO_SNDTIMEO** e **SO_RCVTIMEO**: permettono di cambiare la durata massima di una primitiva di send o receive dopo cui il processo viene sbloccato;
- **SO_SNDBUF** e **SO_RCVBUF**: permettono di cambiare la dimensione del buffer di trasmissione o ricezione, eventualmente eliminando attese per i messaggi di dimensioni elevate (la dimensione massima è di 64 kB);
- **SO_KEEPALIVE**: permette di controllare periodicamente la connessione: il protocollo di trasporto invia messaggi di controllo periodici per analizzarne lo stato;
- **SO_REUSEADDR**: permette di modificare il comportamento della bind(). Il sistema tende a non ammettere più di un utilizzatore alla volta di una porta, infatti ogni ulteriore utilizzatore viene bloccato; con questa opzione, si richiede che la socket sia senza controllo dell'unicità di associazione. In particolare, è utile quando si deve riavviare un server in seguito a un crash e deve essere operativo immediatamente.
- **SO_LINGER**: permette di modificare il comportamento della primitiva close(). A default, il sistema tende a mantenere dopo la close() la memoria in uscita anche per un lungo intervallo di tempo; con questa opzione è possibile modificare l'intervallo.

4.8. Modalità asincrone e non bloccanti per le primitive

Si possono rendere le socket asincrone mediante le primitive **ioctl()** e **fcntl()**. Tramite queste primitive si ottengono socket asincrone che permettono operazioni senza attesa e l'utente viene avvisato del completamento di un'azione con un segnale ad hoc: SIGIO. Questo segnale notifica un cambiamento di stato della socket (causato dall'arrivo di dati) e viene ignorato dai processi che non hanno definito un gestore. Nel caso in cui si debba mandare il segnale a un gruppo di processi, si utilizza la primitiva ioctl() con l'attributo SIOCSPGRP e un flag: se il valore del flag è positivo il segnale arriva a tutti i processi del process group, se è negativo arriva solo al processo che ha il PID uguale al valore del flag negato.

Per ottenere socket non bloccanti invece si deve usare la primitiva ioctl() insieme al parametro FIONBIO con valore 1 per il non blocco (di default, bloccante, è a 0). In questo modo si modificano le primitive:

- accept() restituisce un errore di tipo EWOULDBLOCK;
- connect() restituisce un errore di tipo EINPROGRESS;
- recv() e read() restituiscono un errore di tipo EWOULDBLOCK;
- send() e write() restituiscono un errore di tipo EWOULDBLOCK.

Rendendo una socket asincrona o asincrona non bloccante ci si trova in una situazione di uso in cui le operazioni vengono richieste ma il processo non si sospende in attesa della terminazione. Nel caso di azioni di lettura serve a poco dato che ci interessa il dato in arrivo, ma in caso di scrittura possiamo comandare le azioni e non aspettare il termine: in caso di scrittura non bloccante, si può assumere che l'operazione sia passata al livello sottostante del driver e si può assumere che abbia successo, anche se non si sa quando.

4.9. La primitiva select

La select() permette di gestire l'attesa multipla sincrona. È una primitiva con time-out intrinseco; le azioni di comunicazione su socket sono potenzialmente sospensive. La select() invocata sospende il processo fino al primo evento o fino al time-out. Gli eventi:

- di lettura rendono possibile e non bloccante un'operazione: quando sono presenti dati da leggere tramite la recv(), quando in una socket passiva c'è una richiesta da accettare tramite la connect() e quando in una socket connessa si è verificato un EOF o un errore;
- di scrittura segnalano un'operazione completata: quando in una socket la connessione è completata tramite la connect(), quando si possono spedire altri dati con la send() e quando in una socket connessa il pari ha chiuso o si riceve un errore;
- anomali ed eccezionali segnalano un errore o un'urgenza: quando arrivano dati out-of-band e quando le socket sono inutilizzabili, quindi è stata fatta una close() o una shutdown().

Il corpo della select è **int select(nfds, readfds, writefds, exceptfds, timeout)**; in particolare i tipi sono:

- size_t nfds;
- int* readfds;
- int* writefds;
- int* exceptfds;
- const struct timeval* timeout.

La select invocata richiede al sistema operativo di passare in output informazioni sullo stato interno di comunicazione: all'invocazione, segnala nelle maschere gli eventi di interesse e il tempo; al completamento, restituisce il numero di eventi occorsi e indica quali con le maschere rimaste e il tempo. Inserendo un time-out pari a NULL, l'azione è sospensiva bloccante sincrona e si attende per sempre (31 giorni); inserendo invece 0 come valore l'azione è non bloccante, passante e si lavora a polling dei canali. La chiamata esamina gli eventi per i file descriptor specificati nelle tre maschere relative alle tre tipologie; i bit della maschera corrispondono ai file descriptor a partire dal 0 fino a quello dato come primo parametro. Al ritorno della chiamata alla select le maschere sono modificate in relazione agli eventi per i corrispondenti file descriptor: 1 se l'evento si è verificato, 0 altrimenti. Anche un solo evento di lettura/scrittura/anomalo termina la primitiva select, dopo cui si possono o trattare tutti o uno solo, anche selezionando un qualunque ordine del servizio. Per facilitare l'usabilità si introducono operazioni sulle maschere, che sono array di bit, di dimensione diversa per le diverse architetture.
È possibile gestire la concorrenza anche dal lato client: il client può usare la select e politiche di servizio opportune per gestire più interazioni e l'asincronismo oppure può generare più processi e far gestire a ciascuno una diversa interazione con il server: questo permette di interagire anche con più server contemporaneamente.

5. Servizi applicativi standard in Internet

5.1. Introduzione

La diffusione di Internet ha avuto luogo grazie alla sua proposta di servizi da diffondere agli utenti già agganciati e creando un'offerta che potesse invogliare nuovi altri utenti; le applicazioni sono costruite sui protocolli con interfacce locali diverse. Tra le applicazioni che sono state standardizzate ci sono due macrocategorie:

- **applicazioni client-server**: sono applicazioni punto a punto che coinvolgono due entità alla volta, come ad esempio la stampa remota, il terminale remoto, che permette l'accesso a nodi remoti, e il trasferimento di file, che permette di trasferire file tra nodi diversi. Proprietà fondamentali dell'implementazione sono la trasparenza dell'allocazione (o meno), il modello C/S, che può avere stato o meno, e la standardizzazione;
- **applicazioni di infrastruttura distribuita**: sono applicazioni che prevedono un'infrastruttura sempre presente per fornire un servizio, come ad esempio i sistemi di nomi globali per Internet, le mail, le news, etc.

Alcune applicazioni sono disponibili solo per il sistema operativo UNIX, come i servizi remoti **rsh**, **rwho**, **rlogin**, etc. Altre applicazioni invece, indipendenti dal sistema operativo, sono:

- **Telnet**, *Virtual Terminal Protocol*;
- **FTP**, *File Transfer Protocol*;
- **TFTP**, *Trivial File Transfer Protocol*;
- **SMTP**, *Simple Mail Transfer Protocol*;
- **NNTP**, *Network News System Transfer Protocol*;
- **LPD**, *Line Printer Daemon Protocol*;
- **DNS**, *Domain Name System*;
- **NNTP**, **Gopher**, **HTTP**, etc.: applicazioni per la diffusione della conoscenza con più o meno trasparenza.

5.2. Telnet – *Virtual Terminal Protocol*

Telnet serve per gestire l'eterogeneità di sistemi operativi e hardware: il terminale locale diventa un terminale sul/del sistema operativo remoto. È un'applicazione standard per i sistemi con TCP/IP mentre per i sistemi UNIX BSD si usa rlogin (remote login). Telnet è un protocollo eterogeneo, infatti è costruito su connessioni TCP/IP, la connessione con un server remoto avviene tramite TCP e ci si può agganciare con il modello C/S a qualunque server. Le caratteristiche sono:

- la comunicazione è simmetrica, con funzioni distinte, complementari e differenziate;
- la gestione dell'eterogeneità avviene tramite l'interfaccia di terminale virtuale;
- la negoziazione delle opzioni del collegamento tra il client ed il server (ASCII a 7/8 bit, con bit di controllo, etc.).

Nel modello C/S di Telnet, il client è un processo che svolge due attività principali:

- stabilisce una connessione TCP con il server, accetta i caratteri dall'utente e li invia al server;
- accetta i caratteri che riceve in risposta dal server e li visualizza sul terminale locale dell'utente.

Il server invece è un processo che deve sia accettare la richiesta di connessione del client, eseguire le azioni richieste e mandare il risultato al client, sia continuare a ricevere richieste: per questo viene creato un demone sul server e viene generato un processo figlio per la gestione di ogni singola sessione. Per collegarsi ad un server Telnet occorre il nome logico dell'host o il suo indirizzo fisico IP e il numero della porta da contattare, che di default è 23; il controllo del flusso, di default, viene fatto dal server.
L'esigenza di un terminale virtuale è sentita nelle reti e in particolare nell'internetworking, infatti Telnet nasce per risolvere l'eterogeneità dei terminali, che possono differire gli uni dagli altri per:

- il set di caratteri;
- la diversa codifica dei caratteri;
- la lunghezza della linea e della pagina;
- i tasti funzioni individuati da diverse sequenze di caratteri (escape sequence).

Sulla rete si considera un unico terminale standard e in corrispondenza di ogni stazione di lavoro si effettua la conversione da terminale locale a terminale standard virtuale e viceversa: Telnet, così come FTP, si basa su un modello di terminale detto NVT, *Network Virtual Terminal*. Telnet è half-duplex e presenta diversi problemi: se viene usato in modalità *one char at a time*, inviando quindi un carattere alla volta, sorgono problemi di overhead; se viene invece usato in modalità *one line at a time* sorgono problemi di buffering, causando problemi di ritardo nei dati linemode. Nel caso in cui lo stream dei dati sia pieno, lo stream utilizza i dati urgenti per comunicare e sono riconosciuti i seguenti comandi:

- flush: scarta tutto;
- no client flow control: il client non fa più un controllo del flusso;
- client flow control sliding window: cambiamento della dimensione e controllo del client.

5.3. NVT – *Network Virtual Terminal*

È un modello standard per la comunicazione dei terminali. All'avvio, NVT prevede l'uso di caratteri con rappresentazione a 7 bit USASCII (caratteri normali da 1 byte con il bit alto a 0, l'ultimo bit è per il reset); in seguito, la rappresentazione della sessione NVT può anche cambiare, se negoziata opportunamente tra i pari. In generale, per ogni informazione scambiata, il client converte i caratteri utente nel formato NVT prima di inviarli al server; i dati viaggiano in formato standard e una volta ricevuti dal server, questo li trasforma nel suo formato locale (e viceversa al ritorno). La connessione è unica e viene effettuato il controllo in banda: è presente l'invio di caratteri di controllo e funzioni di controllo insieme ai dati normali. È presente anche l'opzione di negoziazione del terminale, quindi si può negoziare la connessione sia all'inizio sia successivamente per selezionare le opzioni telnet; per richiedere la terminazione dell'applicazione è definito un tasto di interruzione concettuale.

5.4. rlogin – *Remote Login*

È un servizio di login remoto su un'altra macchina UNIX. Se l'utente ha una home directory in remoto, accede a quel direttorio, altrimenti entra nella radice della macchina remota. Il servizio supporta il concetto di *trusted hosts* per garantire corrispondenze tra utenti (si può quindi usare senza password) mentre, in genere, il superutente non può passare da una macchina ad un'altra per problemi di sicurezza. Rlogin conosce l'ambiente di partenza e quello di arrivo, ha nozione di stdin, stdout e stderr (collegati al client mediante TCP), esporta l'ambiente del client verso il server, utilizza più processi (due per parte) e lavora un carattere alla volta. Rlogin è molto più snello di Telnet ma con prestazioni limitate e non ottimizzate, infatti il suo codice è di migliaia di righe mentre quello di Telnet è di decine di migliaia. Vengono utilizzate due connessioni, quattro processi ed è presente il controllo del flusso: il client rlogin tratta localmente i caratteri di controllo del terminale. A livello di implementazione, è presente un client rlogin e un server remoto rlogind: il client crea una connessione TCP con il server rlogind e poi divide le funzioni di input e output:

- il genitore gestisce i caratteri che vanno dal client al server remoto;
- il figlio gestisce i caratteri in arrivo sul client dal server remoto.

Il server si collega ad uno shell remoto con coppia master-slave di uno pseudo-terminale (dopo aver lasciato libero il demone rlogind).

5.5. FTP – *File Transfer Protocol*

È il protocollo usato per trasferire file. FTP adopera TCP, affidabile ed orientato alla connessione, mentre TFTP si basa su UDP essendo più semplice e con meno possibilità. È presente il controllo dell'identità, quindi si accede con una password; l'esecuzione è a livello applicativo o con accesso interattivo utente (si può richiedere la lista dei file di un direttorio remoto, creare un direttorio remoto, etc.) ed è presente la specifica differenziata del formato dei dati (rappresentazione): file di testo o file binari.
I comandi per trasferire file sono:

- **put local-file** [**remote-file**]: memorizza un file locale sulla macchina remota;
- **get remote-file** [**local-file**]: trasferisce un file remoto sul disco locale;
- **mget** e **mput**: utilizzano metacaratteri nei nomi dei file per trasferire gruppi di file.

Esistono nodi server di FTP che sono contenitori di informazioni a cui si può accedere liberamente: si usa *FTP anonymous* verso i server e si possono scaricare liberamente i file. In FTP tutte le informazioni viaggiano in chiaro e si usa una codifica numerica a tre cifre: la prima cifra codifica le interazioni, la seconda codifica le risposte e la terza specifica più precisamente alcuni dettagli. L'accesso da parte di più client è concorrente e sono presenti almeno due collegamenti per ogni client e per ogni server: una connessione di controllo, sempre viva e posizionata sulla porta 21 del server, e una (o più) di dati (ciascuna attiva solo per ogni trasferimento di file) posizionata sulla porta 20 del server; sono quindi impiegati più processi a parte il demone iniziale. Dato che si utilizza TCP, gli end-point individuano una connessione unica ed è sufficiente un end-point diverso per avere una connessione diversa. FTP offre anche la possibilità di avere uno stato: in caso di trasferimento di grandi moli di dati, se ci si blocca, non si deve ripartire dall'inizio ma dall'ultima posizione trasferita. All'avvio della connessione dati, chi esegue la connect è detto attivo e l'altro passivo:

- nel caso in cui il client sia attivo, esso esegue la connect e il server deve aver già fatto la listen ed essersi messo in attesa di richieste da parte dei client: in questo caso il client deve intervenire garantendo che le azioni siano nell'ordine giusto e coordinarsi con il server (che deve mandare un evento di pronto);
- nel caso in cui invece il server sia attivo, il client esegue la listen e fa una accept sulla sua porta, poi utilizza la get o la put. Il server deve solo fare una connect sulla porta del client: in questo caso il client può facilmente intervenire producendo le azioni nell'ordine necessario (senza overhead).

5.6. Posta elettronica

La posta elettronica permette lo scambio di messaggi tra utenti, così come nel servizio postale, in modo asincrono. Questa è una caratteristica fondamentale, infatti a differenza di Telnet e FTP il mittente non aspetta il destinatario. I messaggi possono essere dei semplici testi oppure dei file interi (si può fare un uso alternativo a FTP). Di norma, si interagisce con le mail attraverso dei programmi che le leggono, ma questi non c'entrano con il protocollo di mail (RFC 822): il formato dei dati iniziale della mail, uno standard dei primi di Internet, è molto semplificato ed è legacy. I messaggi sono di testo puro e sono divisi in header (descrittore) e corpo (contenuto del messaggio). Il testo dei messaggi nello standard RFC 822 è in formato ASCII puro. Lo standard definisce anche le funzioni del supporto che specifica come deve essere trattato il messaggio. Grazie a MIME, *Multipurpose Interchange Mail Extension*, è possibile inserire messaggi con formati diversi in un unico corpo di un messaggio che il protocollo riconosce automaticamente. Il servizio di mail usa un sistema di comunicazione punto a punto attraverso una rete di UA, *User Agent*, e MTA, *Mail Transfer Agent* che sono indipendenti tra loro: gli MTA trasferiscono le mail dallo UA sorgente a quello di destinazione. In questo sistema di comunicazione gli UA sono gli utenti finali e gli MTA sono gli agenti che si occupano del trasporto. Un processo UA in background diventa il client dell'attività MTA, che:

- mappa il nome della destinazione in indirizzo IP o di intermediario;
- tenta la connessione TCP con il mail server successivo o di destinazione;
- se la connessione è riuscita, copia il messaggio al successivo passo.

Per il routing tra MTA, il sistema di nomi della posta elettronica può basarsi su un sistema di nomi di DNS o basarsi su altri cammini e percorsi; i diversi MTA possono organizzarsi anche in modo del tutto indipendente dalle normali forme di routing di IP. L'accesso finale ai singoli messaggi da parte dell'utente UA è regolato da diversi strumenti e protocolli quali POP, *Post Office Protocol*, IMAP, *Internet Mail Access Protocol*, etc.

5.7. SMTP – *Simple Mail Transfer Protocol*

È il protocollo standard per il trasferimento della mail tra mailer (MTA) che si connettono e scambiano messaggi di posta in chiaro; gli scambi di messaggi sono codificati tra un client ed un server. I ruoli tra sender e receiver (o client e server) possono essere invertiti per trasmettere la posta diretta nel verso opposto. I comandi sono parole composte da caratteri ASCII mentre le risposte sono composte da un codice numerico, di tre cifre, e testo. Così come in FTP, nelle risposte la prima cifra codifica le interazioni, la seconda codifica le risposte e la terza specifica più precisamente alcuni dettagli.

5.8. Usenet news

Si parla di news come un insieme di gruppi di discussione collettiva; ogni gruppo riguarda un particolare argomento (topic) e permette di partecipare alla discussione su tale argomento, scambiando informazioni e facendo domande, ricevendo risposte, etc., su insiemi aperti di interessi pubblici. L'architettura è formata da tanti server in cui sono mantenute le news e da molti lettori di news (i client): il client collegandosi ad un server ottiene le news presenti su quel nodo. I client sono strumenti per l'accesso applicativo alle news e consentono anche di inviare news ai gruppi di interesse. Gli agenti si coordinano usando la porta 119 e usano una connessione TCP. La comunicazione tra client e server è uguale a quella in SMTP.

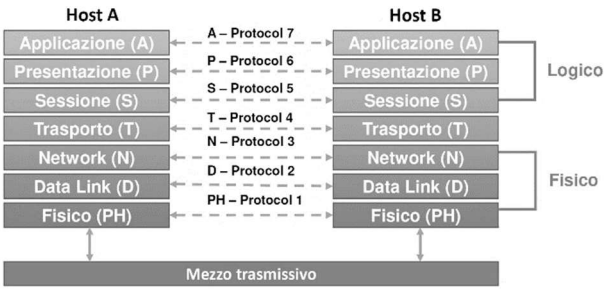
6. OSI – *Open System Interconnection*

6.1. Introduzione

OSI è uno standard di comunicazione tra sistemi aperti che permette a sistemi eterogenei di interoperare, ossia consente di comunicare e operare tra loro in modo aperto. Un importante obiettivo laterale per i provider di comunicazione è la gestione dei sistemi da remoto e OSI è uno standard per la gestione dei sistemi remoti: si occupa del controllo, del coordinamento e del monitoraggio di sistemi interconnessi eterogenei per consentirne una gestione efficiente e a distanza senza porre limiti di località e di co-residenza.
OSI nasce con obiettivi di razionalizzazione per qualunque tipo di comunicazione tra sistemi diversi, infatti protocolli e architetture di rete diverse proprietarie non erano in grado di interagire in modo sistematico: OSI propone quindi standard e schemi di progetto astratti, per razionalizzare, inquadrare e abilitare ogni comunicazione (e guidare le soluzioni). OSI è:

- organizzato a livelli, ciascuno con precisi obiettivi e significati;
- interamente ad oggetti;
- astratto e senza legami con le realizzazioni (proprietarie o meno);
- uno scenario ampio e di riferimento per le soluzioni.

6.2. Architettura



OSI propone un'architettura di soluzione per arrivare a descrivere una comunicazione complessa. L'architettura si basa sul principio di astrazione, che richiede di nascondere i dettagli e mostrare solo le entità utili per l'utente finale. È possibile risolvere problemi complessi separando gli ambiti in modo ordinato e ben identificato attraverso una serie di astrazioni, ovvero i livelli, in modo da decomporre il problema e affrontare le complessità separatamente. OSI prevede sette livelli, uno sopra l'altro: fisico, data link (o collegamento), network (o rete), trasporto, sessione, presentazione e applicazione; i primi tre livelli rientrano nella parte fisica dell'architettura, gli ultimi tre livelli rientrano nella parte logica dell'architettura e il livello di trasporto le separa. Ogni livello ha l'obiettivo di comunicare con il pari e lo fa tramite un protocollo. L'obiettivo di un sistema a livelli è quello di fornire una miglior astrazione; ogni livello inferiore nasconde al livello superiore dei dettagli e ogni livello superiore usa gli strumenti esposti da quello inferiore per il suo obiettivo (e così via). In generale, i livelli OSI sono prescrittivi e non è possibile non considerarli o bypassarli, ma si devono attraversare in modo ordinato, solo secondo la gerarchia introdotta. Ogni livello prevede quindi un protocollo da realizzare e un servizio da presentare/richiedere in modo verticale tramite un'interfaccia.

6.3. Nomi ed entità

In genere, durante l'invio di informazioni, si ha un mittente, ovvero l'entità che ha la responsabilità di iniziare la comunicazione, un ricevente, ovvero chi accetta la comunicazione e poi la sostiene, e gli intermediari, eventuali punti intermedi che devono partecipare alla comunicazione per sostenerla. Il mittente manda dei dati ad un ricevente che può anche rispondere all'invio con un'azione applicativa conseguente. Ogni azione comporta una comunicazione che passa attraverso i livelli, dall'applicazione al fisico, del mittente e del ricevente e almeno fino al livello di rete per gli intermediari. Considerando macchine host e anche nodi intermedi di rete, ogni elemento attivo in un livello viene detto entità. Si

definisce un sistema di nomi per le entità usando la lettera iniziale maiuscola del livello a cui appartengono, ad esempio S-layer e S-protocol per il livello di sessione; i nomi non sono usati solo per i livelli e i protocolli ma anche per le necessarie entità che fanno parte dell'implementazione. L'interfaccia logica tra una n-1 entità e una n entità è detta SAP, *Service Access Point*, la quale identifica il punto di accesso che un servizio OSI offre al suo livello superiore; ogni SAP deve avere un nome unico per essere identificato e possono essere presenti anche più SAP per uno stesso livello. Per identificare un'entità si devono nominare tutti i SAP di ogni livello fino al livello 1.

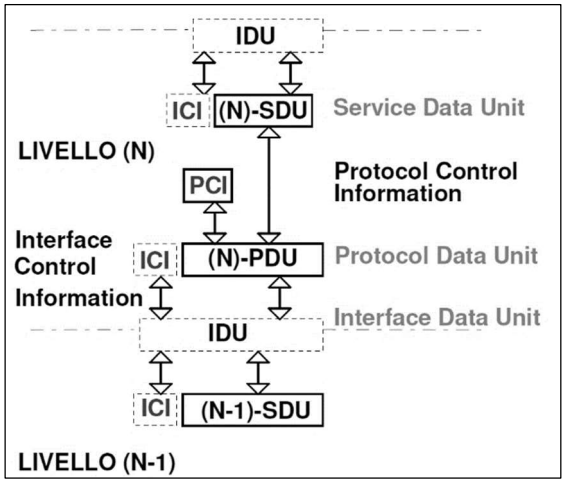
6.4. Protocolli e implementazioni

Per quanto riguarda i protocolli, OSI definisce solamente le specifiche di comunicazione senza dettare nessuna specifica a livello locale né suggerire alcuna tecnologia di soluzione; l'organizzazione a SAP è astratta e potrebbe essere applicata alla modellazione anche di molti altri sistemi. Per ogni livello sono possibili e riconosciute implementazioni a procedure, a processi e ancora più parallele: nello standard OSI non si citano mai processi e procedure ma si usano termini neutri, come attività e flusso di informazioni in modo da restare il più possibile generale.

6.5. Formato dei messaggi

I messaggi scambiati tra i diversi livelli sono standardizzati sia considerando i messaggi scambiati per chiedere un servizio ad un SAP, sia i messaggi scambiati nella realizzazione di un protocollo; le entità che descrivono questi messaggi sono:

- **SDU**, *Service Data Unit*: sono i dati per richiedere i servizi;
- **IDU**, *Interface Data Unit*: sono le richieste portate all'interfaccia;
- **PDU**, *Protocol Data Unit*: realizza le operazioni in orizzontale.



Le operazioni che vanno verso il basso sono incapsulate e devono passare informazioni e comandi al protocollo, per poi essere trattate dal protocollo in modo da definire come agire. All'interfaccia con il livello sottostante, l'IDU contiene anche la parte specifica per il protocollo sottostante, detta ICI, *Interface Control Information*, che coordina le operazioni, determinando il protocollo e definendo poi un PCI, *Protocol Control Information*, per la realizzazione del protocollo. Il PDU viene formato aggiungendo informazioni al dato passato tra entità pari.

6.6. Modalità di connessione

Ci sono due modalità di connessione:

- **connectionless**: ogni unità di dati è trasferita in modo indipendente dalle altre unità e senza ordine; non è offerta nessuna qualità del servizio, nessuna negoziazione e lo scambio di informazioni tra i due pari avviene senza storia. È adatta per lo scambio di dati occasionali e senza qualità della comunicazione, infatti un messaggio mandato dopo può arrivare prima di un precedente;
- **connection-oriented**: si stabilisce una connessione tra entità pari che devono comunicare, con caratteristiche della connessione negoziate durante la fase iniziale, e sono supportati messaggi multipli. In questa modalità la comunicazione tra due utenti di pari livello avviene in tre fasi: apertura della connessione, trasferimento di dati e terminazione della connessione. Il servizio

connection-oriented di un livello deve fornire le opportune funzionalità per le tre fasi con l'opportuna qualità del servizio.

6.7. Primitive

Le entità pari cooperano tramite primitive per implementare le funzionalità del livello a cui appartengono. Le primitive base sono:

- **data**: serve per trasmettere il contenuto;
- **connect**: serve per aprire una connessione;
- **disconnect**: serve per chiudere una connessione.

Ciascuna primitiva ha quattro forme:

- **request**: il service user richiede un servizio;
- **indication**: il service provider indica al service user che è stato richiesto un servizio;
- **response**: il service user specifica la risposta alla richiesta di servizio;
- **confirm**: il service provider segnala la risposta alla richiesta di servizio.

Nel caso detto *sincrono* possono essere presenti anche tutte le forme. Nel dialogo tra pari, si può utilizzare la forma "nome primitiva, punto, tipo primitiva", come ad esempio *S-connect.response*. Le primitive possono essere asincrone nel caso in cui non si dà nessuna conferma al client, sincrone nel caso in cui si dà il risultato al client con conferma richiedendo un'azione al server e asincrone bloccanti nel caso in cui si dà solo la conferma al client.

6.8. Livello network

Dato che non è possibile controllare direttamente il cammino da un qualunque mittente ad un qualunque destinatario, il livello di network si occupa delle diverse realizzazioni di routing tra reti diverse, oltre a definire il sistema di nomi delle entità; l'obiettivo è il passaggio delle informazioni interferendo il meno possibile sul comportamento locale. I compiti del livello di network sono l'indirizzamento, il controllo di flusso tra due pari e il controllo di congestione nell'intero sistema. Gli obiettivi sono il migliorare l'efficienza ed evitare ingiustizia e deadlock. Un principio che rispetta è quello di separazione: i nodi intermedi devono poter interagire solo per le funzionalità necessarie e non essere toccati ai livelli applicativi.

6.9. Livello trasporto

Il trasporto separa i livelli applicativi da quelli fisici; questo livello comincia a considerare la struttura dei nodi partecipanti, in particolare gli end-point. Ogni livello ha le proprie entità e SAP; focalizzando l'attenzione solo sui livelli di network e trasporto, un nodo potrebbe avere molte T-SAP di trasporto e una sola N-SAP di rete: in questo caso le applicazioni dovrebbero specificare quali enti sono coinvolti per ogni comunicazione. Lo stesso discorso vale per ogni livello superiore: un pari che vuole comunicare deve specificare tutta la pila di SAP proprie e anche quelle che permettono di arrivare all'altro in modo non ambiguo; anche gli intermedi devono essere considerati per la connessione con qualità e devono essere negoziati. La primitiva connect mette in gioco molte entità su tutti i nodi interessati alla connessione e mantiene le risorse impegnate. Il trasporto può spezzare un dato e ricomporlo dopo averlo portato, suddiviso, fino al pari; può quindi lavorare unendo o decomponendo flussi di trasporto rispetto a quelli di rete. L'obiettivo di questo livello è quello di spedire dati sul canale di connessione con correttezza, con certi tempi di risposta e con una certa qualità di servizio.

6.10. Livello sessione

Sul trasporto, che lavora da nodo a nodo, la prima esigenza è il supporto al dialogo. La sessione considera e determina i meccanismi per il dialogo tra entità diverse, tenendo in conto anche due pari che comunicano in modo vario ed eterogeneo. Nella sessione, il dialogo può:

- essere bidirezionale;
- essere molteplice e strutturato in attività separate e diverse;
- considerare le risorse impegnate;
- avere garanzie di correttezza e affidabilità.

Un dialogo può avere molte dimensioni possibili (scambio di testo, scambio di file, video in real-time, etc.) e anche molte specifiche differenziate di qualità (scaricamento di molti frame alla volta, senza ripartire da zero, garanzia di persistenza, dialogo uno alla volta, etc.). Il dialogo trae vantaggio da un supporto ad hoc di funzioni che siano capaci di garantire i protocolli e i meccanismi per il miglior supporto. La sessione standardizza una serie di componenti per avere a disposizione tutte le funzionalità necessarie secondo i requisiti; in particolare i requisiti sono specializzati per la gestione dell'interazione (modalità di dialogo half-duplex, full-duplex o simplex), per la sincronizzazione, inserendo dei punti di sincronizzazione, e per la gestione delle eccezioni.

Il livello di sessione coordina il dialogo basandosi sul servizio offerto organizzato in unità funzionali, ognuna legata ad un insieme di primitive e parametri; il numero delle unità funzionali cresce per i livelli verso l'applicazione. Il servizio di sessione offre 58 primitive raggruppate in 14 unità funzionali. Il dialogo è diviso in attività indipendenti che si possono gestire (iniziare, sospendere, cancellare, etc.) ed è possibile notificare eccezioni ai servizi corrispondenti. Ogni pari può richiedere il livello di servizio adatto alle esigenze sue e dell'invocazione. Nel dialogo tra pari c'è la possibilità di intervenire automaticamente:

- se la trasmissione di un file da due ore si blocca dopo un'ora, riprendendo dal risultato del trasferimento precedente;
- se si verificano errori nella comunicazione, si ha roll-back;
- se nel trasferimento di molti MB avviene un crash, ricominciando.

I punti di sincronizzazione sono punti per introdurre checkpoint nel dialogo e consentire di articolarlo in modo organizzato. I tipi previsti sono due:

- **punti di sincronizzazione maggiore**: il mittente attende, in modo sincrono bloccante, che il ricevente confermi;
- **punti di sincronizzazione minore**: si invia, con conferma o meno, e il mittente può continuare a spedire dati. Il mittente non deve segnalare subito la ricezione di un punto minore e la conferma di un punto minore conferma anche tutti i punti precedenti.

I punti di sincronizzazione sono usati per re-sincronizzarsi verso uno stato definito dai punti stessi in caso di recovery: in caso di punti maggiori, si attende fino ad una confirm; in caso di punti minori, si accumulano ed eventualmente si attende. Tipicamente, si può negoziare il numero di punti di sincronizzazione minore che sono in attesa di conferma: si determina la dimensione di una finestra che scorre (sliding window) di punti non confermati e al riempimento della finestra il mittente deve aspettare la conferma per procedere con altri punti di sincronizzazione. I punti di sincronizzazione di un dialogo possono essere usati nel recovery per ritrovare uno stato significativo in modo automatico o negoziato. In caso di recovery, si prevedono anche strategie diverse; quelle previste consentono:

- l'abbandono: si fa un reset della comunicazione e l'utente può decidere di ripeterla;
- il ripristino: la comunicazione è riportata nello stato precedente, più nello specifico all'ultimo punto di sincronizzazione maggiore confermato;
- il ripristino diretto dall'utente: la comunicazione è riportata in uno stato arbitrario senza controllo delle conferme mancanti di punti di sincronizzazione ed è compito delle applicazioni decidere in modo coordinato uno stato da cui ricominciare la trasmissione.

Strutturazione e sincronizzazione del dialogo avvengono attraverso oggetti astratti detti token, intesi come gettoni di autorizzazione: un solo utente possiede il token in ogni momento e può usare un insieme di servizi del livello sessione. La primitiva connect per stabilire la connessione permette di negoziare anche i token. Ci sono vari tipi di token:

- **data token**: permette di spedire i dati in modalità half-duplex;
- **release token**: permette di richiedere la terminazione;
- **synchronize minor token**: permette di creare un punto di sincronizzazione minore;
- **synchronize major token**: permette di creare un punto di sincronizzazione maggiore.

6.11. Livello presentazione

La codifica delle informazioni non è univoca e ogni pari può usare codifiche diverse. Il livello di presentazione offre i servizi per consentire una corretta interoperabilità sui dati, ad esempio superare il problema della codifica dei dati dei diversi pari, da mandare e da ricevere. Il livello di presentazione norma gli strumenti e i protocolli necessari per una corretta gestione dei dati eterogenei nei sistemi aperti; la presentazione affronta quindi tutto il problema della rappresentazione dei dati e delle differenze naturali tra i sistemi che comunicano ma anche dei casi di necessità di codifiche ad hoc per compressione dei dati (efficienza) o crittografia (sicurezza). Ogni pari può usare codifiche diverse dipendentemente dall'architettura del sistema, dai sistemi operativi considerati, dai linguaggi, dalle applicazioni specifiche, etc. I dati devono essere scambiati dopo un accordo tra i pari che supera gli eventuali problemi di eterogeneità. In caso di problemi di eterogeneità, ci sono due soluzioni:

- dotare ogni nodo di tutte le funzioni di conversione possibili per ogni possibile rappresentazione dei dati: si ha un'elevata performance ma servono $n(n-1)$ funzioni di conversione;
- concordare un formato comune di rappresentazione dei dati; ogni nodo possiede le funzioni di conversione da e per questo formato: si deve implementare un minor numero di funzioni di conversione, pari a $2n$. Dato che questa opzione richiede un numero molto inferiore di funzioni, si usa sempre e solamente questa.

Nel caso in cui c'è una completa uniformità del formato dati, non si fanno trasformazioni, ad esempio se si usa la JVM di Java. Nel caso in cui non si ha solo eterogeneità dei dati ma anche una situazione dinamica in cui non si sa come dialogare con gli altri, bisogna decidere cosa comunicare e in che modo farlo usando il protocollo stesso: se non c'è accordo su cosa dire, va negoziato un linguaggio comune; se non c'è accordo ma si sa cosa dire, bisogna solo accordarsi sul formato dei dati standard e della trasformazione specifica dei contenuti di interesse usando un linguaggio di descrizione dei dati. I dati possono viaggiare così come sono, quindi come dei valori, offrendo un'elevata efficienza, o anche con dei descrittori del dato che producono ridondanza e affidabilità; si possono usare diversi gradi di ridondanza a seconda del costo associato alla comunicazione (impegno di banda). Il livello di presentazione stabilisce come negoziare e definire una base comune per la comunicazione; spesso l'accordo può essere ottenuto solo con protocolli detti di negoziazione, ossia a molte fasi, con durata non predefinita e anche lunga nei casi peggiori. Il numero di fasi spesso dipende dal numero di eventi che si sono verificati durante il protocollo stesso. Il **bidding** è l'insieme delle fasi necessarie per raggiungere un accordo e comprende, normalmente:

1. il sender fa un broadcast della propria esigenza;
2. i receiver fanno un'offerta (bid);
3. il sender sceglie tra i bid dei receiver;
4. il receiver accoglie l'ok definitivo (contract);
5. accordo.

Non ci sono prenotazioni delle fasi: nella fase 4 il receiver può rifiutare e si riparte dalla fase 3 (o nel peggiore dei casi dalla fase 1).

Il livello di presentazione definisce un linguaggio astratto di specifica noto come ASN.1, *Abstract Syntax Notation*, per casi dinamici, e un linguaggio concreto di descrizione noto come BER, *Basic Encoding Rules*, usato estensivamente. In un caso generale si può aver bisogno di ASN.1 per accordarsi su cosa si vuole comunicare e fare una negoziazione e di BER per stabilire quale sia il formato comune dei dati da trasferire.

6.12. Livello applicazione

Il livello di applicazione è il livello che si interfaccia con l'utente finale della comunicazione. Il suo obiettivo è l'astrazione: nascondere la complessità dei livelli sottostanti coordinando le applicazioni distribuite. Questo livello definisce un insieme di servizi indipendenti dal sistema ed ambienti standard agli utenti. OSI adotta un approccio basato sul modello a oggetti per la specifica delle applicazioni:

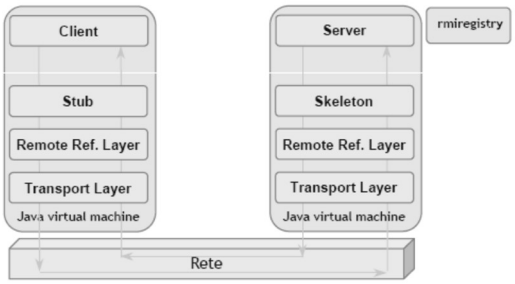
- uso di template e package per definire gli oggetti;
- pura ereditarietà statica tra astrazioni;
- oggetti da manipolare come interfacce.

7. Java RMI – *Remote Method Invocation*

7.1. Introduzione

L'architettura RMI introduce la possibilità di richiedere l'esecuzione di metodi remoti in Java, integrando il tutto con il paradigma a oggetti. RMI è un insieme di strumenti, politiche e meccanismi che permettono ad un'applicazione Java in esecuzione su una macchina di invocare i metodi di un oggetto di un'applicazione Java in esecuzione su una macchina remota; viene creato localmente solo il riferimento ad un oggetto remoto, che è invece effettivamente attivo su un nodo remoto e un programma client invoca i metodi attraverso questo riferimento locale mantenuto in una variabile interfaccia. Le interazioni in RMI sono sempre sincrone e bloccanti; l'ambiente di lavoro è unico dato che si utilizza il linguaggio Java, ma si possono usare sistemi anche eterogenei grazie alla portabilità del codice.

7.2. Architettura RMI



In Java non sono direttamente possibili i riferimenti remoti, ma RMI permette di costruirli tramite due proxy: lo stub dal lato client e lo skeleton dal lato server; tramite questi due proxy, usando quindi il pattern proxy, si nasconde al livello applicativo la natura distribuita dell'applicazione. Tuttavia, il client riuscirà a percepire differenze nell'affidabilità, semantica, durata, etc. Per poter usare i riferimenti remoti, RMI utilizza una variabile interfaccia che dinamicamente può contenere un riferimento a un'istanza di una classe qualunque che implementa l'interfaccia stessa. I livelli presenti nell'architettura RMI sono:

- **stub**: è il proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto;
- **skeleton**: è l'entità remota che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server;
- **RRL**, *Remote Reference Layer*: è responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni di stream-oriented connection;
- **TL**, *Transport Layer*: è il livello di trasporto connesso e:
  - è responsabile della gestione delle connessioni fra i diversi spazi di indirizzamento (JVM diverse);
  - gestisce il ciclo di vita delle connessioni e le attivazioni integrate in JVM;
  - può utilizzare protocolli applicativi diversi, purché siano connection-oriented (di default, in RMI, si usa TCP a livello di trasporto);
  - utilizza un protocollo proprietario.
- **registry**: è il servizio di nomi che consente al server di pubblicare un servizio e al client di recuperarne il proxy.

I due livelli RRL e TL sono parte della macchina virtuale JVM. Nel modello a oggetti distribuito di Java, un oggetto remoto consiste in:

- un oggetto i cui metodi sono invocabili da un'altra JVM, potenzialmente in esecuzione su un differente host;
- un oggetto descritto tramite una o più interfacce remote che dichiarano i metodi accessibili da remoto.

Per quanto riguarda le differenze tra una chiamata remota e una locale, il client invoca un metodo di un oggetto remoto attraverso un riferimento remoto (variabile interfaccia): la sintassi è uguale, quindi c'è trasparenza per il client, ma la semantica, in questo caso at-most-once con uso di TCP, è diversa: le chiamate locali hanno un'affidabilità massima, quelle remote possono fallire. Il server remoto nell'architettura RMI esegue ogni chiamata in modo indipendente e parallelo.

7.3. Interfacce ed implementazione

In RMI si fa distinzione tra:

- definizione del comportamento; per questo si usano le interfacce:
  - ogni interfaccia deve estendere java.rmi.Remote;
  - ogni metodo deve propagare java.rmi.RemoteException.
- implementazione del comportamento; per questo si usano le classi:
  - la classe che implementa il comportamento deve implementare l'interfaccia definita;
  - la classe deve estendere java.rmi.UnicastRemoteObject.

I componenti remoti sono riferiti tramite variabili interfaccia, che possono contenere solo istanze di classi che implementano l'interfaccia. Per utilizzare Java RMI è necessario:

1. definire l'interfaccia e l'implementazione del componente utilizzabile in remoto (ovvero l'interfaccia e il server);
2. compilare l'interfaccia e la classe del punto 1 con il comando java, poi generare stub e skeleton con il comando rmic -vcompat della classe utilizzabile in remoto;
3. pubblicare il servizio nel sistema di nomi, attivare il registry e registrare il servizio (il server deve fare una bind sul registry);
4. ottenere, dal lato client, il riferimento all'oggetto remoto tramite il name service, facendo una lookup sul registry, e compilare il client.

A questo punto l'interazione tra client e server può procedere; si noti che è necessario attivare prima il server e poi il client. Ciascun metodo remoto ha un solo risultato di uscita e nessuno, uno o più parametri di ingresso; i parametri devono essere passati per valore (dati primitivi oppure oggetti Serializable) o per riferimento remoto (oggetti Remote). Il registry è un processo in esecuzione sull'host del server e registra tutti i servizi: invoca tante operazioni di bind/rebind quanti sono gli oggetti server da registrare ciascuno con un nome logico.

7.4. RMI registry

Un RMI registry risponde alla necessità di localizzazione di un servizio: un client in esecuzione su una macchina ha bisogno di localizzare un server, in esecuzione su un'altra macchina, a cui connettersi. Il registry mantiene al suo interno un insieme di coppie {name, reference} in cui name è una stringa arbitraria, del tipo //nomehost:porta/servizio, che indica un servizio e reference indica su quale server si trova tale servizio. Quando si avvia il programma rmiregistry sull'host server si può specificare o meno la porta sulla quale lo si sta utilizzando; di default è la porta 1099. Dato che l'rmiregistry viene attivato in una shell a parte, esso è attivato in una nuova istanza JVM separata dalla JVM del server e anch'esso, per ragioni di economicità di implementazione e di principio, è un server RMI. È anche possibile creare all'interno del codice del server un proprio registry con il metodo **public static Registry createRegistry(int port)**, un metodo della classe LocateRegistry; in questo caso il registry verrà creato nella stessa istanza JVM del server.

7.5. Stub e skeleton

Stub e skeleton rendono possibile la chiamata di un servizio remoto come se fosse locale, agendo da proxy. Sono entrambi generati dal compilatore RMI e l'ambiente Java supporta già direttamente la serializzazione e de-serializzazione. La procedura di comunicazione è:

- il client ottiene un riferimento remoto, attua la chiamata dei metodi tramite lo stub e aspetta il risultato;
- lo stub effettua la serializzazione delle informazioni per la chiamata (ID del metodo, identificazione e argomenti) e invia le informazioni allo skeleton utilizzando le astrazioni messe a disposizione dal RRL;

- lo skeleton effettua la de-serializzazione dei dati ricevuti, invoca la chiamata sull'oggetto che implementa il server (dispatching), effettua la serializzazione del valore di ritorno e lo invia allo stub;
- lo stub effettua la de-serializzazione del valore di ritorno e lo restituisce al client.

7.6. Serializzazione e passaggio dei parametri

In generale, in sistemi RPC, i parametri di ingresso e uscita subiscono una duplice trasformazione per risolvere problemi di rappresentazioni eterogenee (livello 6 di OSI, presentazione):

- **marshalling**: processo di codifica degli argomenti e dei risultati per la trasmissione;
- **unmarshalling**: processo inverso di decodifica di argomenti e risultati ricevuti.

In Java, grazie all'uso del BYTECODE uniforme e standard, non c'è bisogno di queste trasformazioni: i dati vengono semplicemente serializzati e de-serializzati, ossia inseriti in un messaggio lineare, utilizzando le funzionalità offerte direttamente a livello di linguaggio:

- la serializzazione trasforma oggetti complessi in semplici sequenze di byte (anche grafi di oggetti) tramite il metodo **writeObject()** su uno stream di output;
- la de-serializzazione decodifica una sequenza di byte e costruisce una copia dell'oggetto originale tramite il metodo **readObject()** da uno stream di input.

Stub e skeleton utilizzano queste funzionalità per lo scambio dei parametri di ingresso e uscita con l'host remoto. I tipi primitivi sono passati per valore sia nel caso locale che nel caso remoto; gli oggetti nel caso locale sono passati per riferimento, mentre nel caso remoto sono passati per valore (interfaccia Serializable deep copy); gli oggetti remoti sono passati per riferimento remoto tramite l'interfaccia Remote.

7.7. Livello di trasporto: concorrenza e comunicazione

Concorrenza e comunicazione sono aspetti chiave di RMI. Per quanto riguarda la concorrenza, si possono realizzare diverse implementazioni, ma l'implementazione deve prevedere un server parallelo che deve essere thread-safe (uso di attività molteplici) e gestire eventuali problemi di sincronizzazione e mutua esclusione con synchronized. Java RMI usa i thread e ne crea uno per ogni richiesta di servizio; data la politica di generazione dei thread e visto che nello skeleton non c'è la loro generazione, è tutto gestito dalla JVM a livello di trasporto.

Anche nel caso della comunicazione non ci sono molti vincoli e c'è libertà di implementazione, l'unico vincolo è quello di un principio di buon utilizzo delle risorse: se esiste già una connessione fra due JVM si cerca di riutilizzarla. Ci sono molte possibilità ma può esserne implementata solo una tra le seguenti:

- si apre una sola connessione e la si utilizza per servire una richiesta alla volta: si causa una forte sequenzializzazione delle richieste;
- si utilizza la connessione aperta se non ci sono altre invocazioni remote che la stanno utilizzando, altrimenti se ne apre una nuova: questo comporta un maggior impiego di risorse, ma si diminuiscono gli effetti causati dalla sequenzializzazione;
- si utilizza una sola connessione (a livello di trasporto) per servire diverse richieste e su quella si fa del demultiplexing per l'invio delle richieste e la ricezione delle risposte;
- la connessione vive a livello di connessione JVM e permane fino alla chiusura delle JVM.

7.8. Distribuzione delle classi (deployment) e class loading

In RMI, ovviamente, le fasi sono sia di sviluppo che di esecuzione; in un'applicazione RMI è necessario che siano disponibili a run-time gli opportuni file .class nelle località che lo richiedono (per l'esecuzione o per la serializzazione/de-serializzazione). Il server deve poter accedere a:

- interfacce che definiscono il servizio, a tempo di compilazione;
- implementazione del servizio, sempre a tempo di compilazione;

- stub e skeleton delle classi di implementazione, a tempo di esecuzione;
- altre classi utilizzate dal server, a tempo di compilazione o esecuzione.

Il client invece deve poter accedere a:

- interfacce che definiscono il servizio, a tempo di compilazione;
- stub delle classi di implementazione del servizio, a tempo di esecuzione;
- classi del server usate dal client (ad esempio i valori di ritorno), a tempo di compilazione o esecuzione;
- altre classi utilizzate dal client, a tempo di compilazione o esecuzione.

Java definisce un ClassLoader, ovvero un'entità capace di risolvere i problemi di caricamento delle classi dinamicamente e di riferire e trovare le classi ogni volta che ce ne sia necessità, oltre che di rilocarle in memoria. Le classi possono essere caricate sia dal disco locale che dalla rete, con vari gradi di protezione. Java consente di definire una gerarchia di ClassLoader diversi, ciascuno responsabile del caricamento di classi diverse e anche definibili dall'utente. I ClassLoader costituiscono ciascuno una località diversa l'uno dall'altro e non interferiscono tra di loro. Il ClassLoader di RMI, chiamato RMIClassLoader, non è un ClassLoader vero e proprio, ma un componente di supporto RMI che esegue due operazioni fondamentali:

- estrae il campo codebase dal riferimento dell'oggetto remoto;
- usa i codebase classloader per caricare le classi necessarie dalla locazione remota.

7.9. Sicurezza in RMI

In ogni JVM, e per ogni ClassLoader, può essere attivato un Security Manager (e dovrebbe esserlo), un controllore di correttezza e sicurezza di ogni singola operazione per quelle località. Sia il client che il server devono essere lanciati specificando il file con le autorizzazioni, detto file di policy, che viene consultato dal security manager per il controllo dinamico della sicurezza. Per l'esecuzione sicura del codice si richiede l'utilizzo del RMISecurityManager, che effettua il controllo degli accessi alle risorse di sistema e blocca gli accessi non autorizzati. Il security manager viene creato all'interno dell'applicazione RMI, sia dal lato client che dal lato server, se non ce n'è già uno istanziato. Esempi di cose che si possono specificare nel file di policy sono:

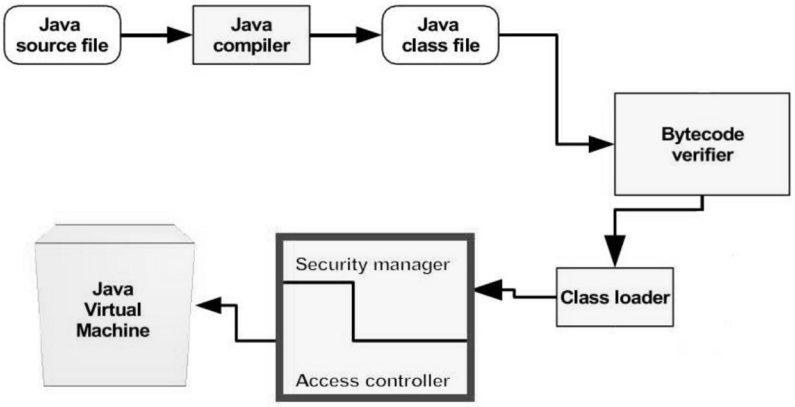
- il range di porte in cui client e server possono connettersi per l'interazione remota;
- consentire di prelevare il codice da un server http;
- consentire di prelevare il codice a partire dalla radice dei direttori consentiti.

Un possibile problema del registry è che accedendoci (è individuabile interrogando tutte le porte di un host) è possibile ridirigere per scopi maliziosi le chiamate ai server RMI registrati: per questo i metodi **bind()**, **rebind()** e **unbind()** sono invocabili solo dall'host su cui è in esecuzione il registry, quindi non si accettano modifiche della struttura C/S da nodi esterni.

Per quanto riguarda il bootstrap, ovvero l'avvio del sistema e il ritrovamento del riferimento remoto, Java mette a disposizione la classe Naming, che realizza dei metodi statici per effettuare le operazioni di registrazione, de-registrazione e reperimento del riferimento del server; i metodi per agire sul registry hanno bisogno dello stub del registry. Per ottenere un'istanza dello stub del registry senza consultare un registry si può costruire localmente un'istanza dello stub a partire dall'indirizzo del server e dalla porta, contenuti nel nome dell'oggetto remoto, e dall'identificatore (locale alla macchina server) dell'oggetto registry fissato a priori dalla specifica RMI della SUN (è una costante fissa).

Nel caso sia necessario ottenere dinamicamente del codice (stub o classi) è necessario localizzare il codice, effettuare il download nel caso sia in remoto, ed eseguire in modo sicuro il codice scaricato. Le informazioni relative a dove reperire il codice sono memorizzate sul server e passate al client by need. Il codebase viene usato dal client per scaricare le classi necessarie relative al server (interfaccia, stub, oggetti restituiti come valori di ritorno).

Riassumendo graficamente il funzionamento di tutto:



8. Sistemi RPC – *Remote Procedure Call*

8.1. Semantiche di comunicazione

Le principali semantiche sono:

- **at-least-once**: questa semantica prevede ritrasmissioni ad intervalli; il messaggio può arrivare anche più volte a causa dei messaggi duplicati dovuti a ritrasmissioni. È una semantica adatta per azioni idempotenti ma in caso di insuccesso non arriva nessuna informazione e può provocare inconsistenza sulle azioni da parte del ricevente. In caso le cose vadano bene il messaggio arriva una volta o anche più volte, in caso vadano male invece il client non sa se il server ha fatto l'azione e il server non sa se il client sa che ha fatto l'azione.
- **at-most-once**: è una semantica in cui i pari lavorano entrambi per ottenere garanzie di reliability: il messaggio, se arriva, arriva al più una volta, mentre in caso di insuccesso non arriva nessuna informazione. Un esempio di progetto at-most-once è TCP. Il client fa ritrasmissioni e il server mantiene uno stato per riconoscere i messaggi già ricevuti, in modo da non eseguire azioni più di una volta. In caso le cose vadano bene il messaggio arriva una volta e viene trattato una volta sola, riconoscendo i duplicati. Se invece le cose vanno male, il client non sa se il server ha eseguito l'azione e il server non sa se il client sa che ha fatto l'azione: manca un coordinamento tra i due.
- **exactly-once**: in questa semantica al termine dell'operazione sia client che server sanno se l'operazione è stata eseguita o meno. I pari lavorano entrambi per ottenere il massimo dell'accordo e della reliability; il messaggio arriva una volta sola oppure entrambi i pari conoscono lo stato finale dell'altro, quindi c'è un accordo completo sull'interazione. È un progetto "tutto o niente": in caso le cose vadano bene il messaggio arriva una volta sola e una sola volta viene trattato, riconoscendo i duplicati; in caso le cose vadano male il client e il server sanno se il messaggio è arrivato o non è arrivato;
- **may-be** (anche detta **best-effort**): in questa semantica il messaggio può arrivare o meno, infatti per limitare i costi si esegue un solo invio di ogni informazione e non vengono eseguite azioni per garantire l'affidabilità. Lo standard sacrifica la QoS all'applicabilità globale. Esempi di progetti may-be sono IP e UDP.

8.2. Introduzione ai sistemi RPC

I sistemi RPC mettono a disposizione, mediante server, delle procedure che possono essere invocate in remoto dai vari client, rendendo quest'invocazione un'estensione del normale meccanismo di chiamata a procedura locale sfruttando il modello C/S nel distribuito. In una chiamata a procedura remota sia i parametri che i risultati viaggiano attraverso la rete, quindi si ha un approccio applicativo di alto livello, livello 7 OSI: il client invia la richiesta ed attende in modo sincrono fino alla risposta fornita dal server. Rispetto alla chiamata a procedura locale:

- sono coinvolti processi distinti su nodi diversi;
- essendo processi distinti, client e server hanno vita separata e non condividono lo spazio di indirizzamento;
- sono possibili malfunzionamenti sui nodi o nell'infrastruttura di comunicazione.

Le proprietà che contraddistinguono RPC sono:

- controllo dinamico del tipo dei parametri e del risultato;
- è presente il trattamento dei parametri di ingresso e uscita dal client al server (e viceversa), detto marshalling, o almeno la serializzazione;
- è trasparente, ovvero opera in remoto ma per il client è come se fosse locale;
- controllo dei tipi e parametrizzazione;
- binding distribuito;

- possibile trattamento degli orfani (ovvero il processo server che non riesce a fornire un risultato e il client senza risposta).

Le RPC non presentano troppo regole standard e ogni sistema usa primitive diverse. Dalla parte del client avvengono delle chiamate di servizio (per il server, per gli argomenti e per il risultato) mentre dalla parte del server sono presenti due tipi di politiche di accettazione e di risposta, con diverse possibilità di concorrenza sul server:

- modalità esplicita e sequenziale: il servizio è svolto da un unico processo che decide quando e se eseguire i metodi;
- modalità implicita e parallela: ogni servizio è eseguito da un processo indipendente, generato in modo automatico per ogni richiesta.

8.3. Tolleranza ai guasti

Per quanto riguarda la tolleranza ai guasti, l'obiettivo applicativo è di mascherare i malfunzionamenti, come:

- la perdita di un messaggio di richiesta o di risposta;
- il crash del nodo del client;
- il crash del nodo del server.

Il client può tentare diverse politiche e diversi comportamenti:

- aspettare per sempre;
- usare un time-out e ritrasmettere (usando identificatori unici);
- usare un time-out e riportare un'eccezione al client.

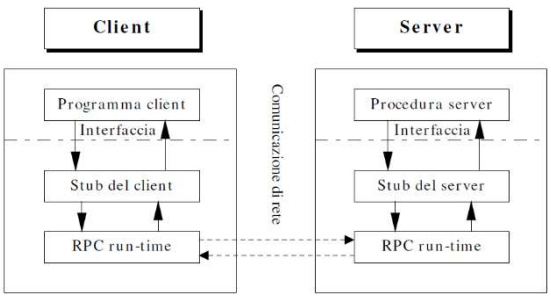
Spesso si assume che le azioni siano idempotenti, ovvero che si possano eseguire un qualunque numero di volte ottenendo sempre lo stesso esito nel modello C/S. Le RPC hanno alcune semantiche tipiche e strategie relative:

- may-be: si usa un time-out per il client;
- at-least-once: si usa un time-out e si ritrasmette;
- at-most-once: si usano tabelle per sapere le azioni effettuate;
- exactly-once: l'azione viene fatta fino alla fine.

Nel caso in cui il crash è avvenuto dal lato client, si devono trattare orfani sul nodo server, ossia i processi in attesa di consegnare il risultato. Le politiche tipiche sono:

- **sterminio**: ogni orfano risultato di un crash viene distrutto;
- **terminazione a tempo**: ogni calcolo ha una scadenza, oltre la quale è automaticamente abortito;
- **reincarnazione** (ad epoche): il tempo è diviso in epoche e tutto ciò che è relativo all'epoca precedente è obsoleto e viene distrutto.

8.4. NCA – *Network Computing Architecture*



Nell'architettura delle RPC sono introdotti ai due end-point di comunicazione degli stub: in questo modo le chiamate appaiono del tutto locali al client e si garantisce la massima trasparenza. Gli stub sono forniti dall'implementazione e vengono generati automaticamente: l'utente finale deve progettare e sviluppare unicamente le reali parti applicative e logiche; tuttavia il loro utilizzo fa sì che servano due chiamate per ogni RPC. Il modello con gli stub è asimmetrico, infatti:

- il client invoca lo stub, che si incarica di tutto: recupero del server, trattamento dei parametri, richiesta al supporto run-time, trasporto della richiesta;

- il server riceve la richiesta dallo stub relativo, che si incarica del trattamento dei parametri dopo aver ricevuto la richiesta pervenuta dal trasporto; al completamento del servizio, lo stub rimanda il risultato al client.

8.5. Passaggio dei parametri e trattamento delle eccezioni

I parametri nelle RPC devono passare tra ambienti diversi; il passaggio dei parametri può essere fatto per valore o per riferimento: in genere si preferisce il passaggio per valore, mentre quello per riferimento richiede un approccio diverso dal solito. Nel caso del passaggio per valore, viene effettuato un passaggio con trasferimento e visita (il valore viene perso sul server); nel caso di passaggio per riferimento, il passaggio è senza trasferimento e l'oggetto viene reso remoto. Vengono quindi usati oggetti che rimangono nel nodo di partenza e devono essere identificati in modo univoco nell'intero sistema; se si vuole riferire un'entità del client, si passa il riferimento alla stessa entità che i nodi remoti possono riferire attraverso RPC. Le RPC hanno previsto il trattamento degli errori tramite una gestione delle eccezioni: gli eventi anomali causati dalla distribuzione o da guasti vengono gestiti tramite una loro integrazione nella gestione locale delle eccezioni e in genere si specifica l'azione per il trattamento anomalo in un opportuno gestore dell'eccezione. È anche possibile inserire la gestione dell'eccezione nello scope del linguaggio, come ad esempio in Java definendo delle Exception.

8.6. IDL - *Interface Definition Language*

Sono linguaggi per la descrizione di operazioni remote, la specifica del servizio (detta firma) e la generazione degli stub. Un IDL deve consentire:

- l'identificazione unica del servizio tra quelli disponibili, usando un nome astratto del servizio e prevedendo versioni diverse del servizio;
- la definizione astratta dei dati da trasmettere in input e output, usando un linguaggio astratto di definizione dei dati (usando interfacce, con operazioni e parametri).

Possibili estensioni degli IDL sono i linguaggi dichiarativi con ereditarietà, ambienti derivati con binder ed altre entità. Gli IDL hanno lo scopo di supporto allo sviluppo dell'applicazione, permettendo di generare automaticamente gli stub dall'interfaccia specificata dall'utente; per tale scopo si usa lo strumento **RPCGEN**, *Remote Procedure Call Generator*, un compilatore di protocollo RPC per generare procedure stub: vengono generati gli stub per il server ed il client da un insieme di costrutti descrittivi per tipi di dati e per le procedure remote in linguaggio RPC.

8.7. Sviluppo e fasi di supporto RPC

Nel caso di un utente che deve utilizzare le RPC, di varia tecnologia, sono presenti molte fasi; alcune sono facilitate e poco visibili, svolte tutte dal supporto, altre invece sono sotto il controllo dell'utente. Dopo la specifica del contratto in IDL, sono presenti le fasi tipiche dell'implementazione:

- compilazione di sorgenti e stub: vengono prodotti gli stub che servono a semplificare il progetto applicativo e ci si assicura che client e server raggiungano un accordo sul servizio da richiedere o fornire;
- binding delle entità: prevede come ottenere l'aggancio corretto tra i client e il server capace di fornire l'operazione;
- trasporto dei dati: è intrinseco allo strumento e si preferisce velocità ed efficienza (sia UDP che TCP);
- controllo della concorrenza: consente di usare gli stessi strumenti per funzioni diverse, con maggiore asincronicità e maggior complessità (ripetizione, condivisione della connessione, processo);
- supporto alla rappresentazione dei dati: per superare l'eterogeneità si trasformano i dati.

8.8. RPC binding e sistemi di nomi

Come detto prima, il binding prevede come ottenere l'aggancio corretto tra i client e il server capace di fornire l'operazione richiesta; il binding del client al server può essere effettuato secondo due possibili modalità:

- **pessimistica e statica**: la compilazione risolve ogni problema prima dell'esecuzione e forza un binding statico, nel distribuito, a costo limitato ma poco flessibile;
- **ottimistica e dinamica**: il binding dinamico ritarda la decisione alla necessità e ha costi maggiori, ma consente di dirigere le richieste sul gestore più scarico, o presente, in caso di sistema dinamico.

Il binding dinamico viene ottenuto distinguendo due fasi nella relazione C/S:

- **fase statica di servizio**, prima dell'esecuzione: il client specifica a chi vuole essere connesso, con un nome unico identificativo del servizio (naming) e si associano dei nomi unici di sistema alle operazioni o alle interfacce astratte; successivamente si attua il binding con l'interfaccia specifica di servizio;
- **fase dinamica di indirizzamento all'uso**: durante l'esecuzione il client deve essere realmente collegato al server che fornisce il servizio richiesto al momento dell'invocazione (addressing), quindi in questa fase si cercano gli eventuali server pronti per il servizio richiesto (usando sistemi di nomi che sono stati introdotti proprio a tale scopo).

La fase statica di servizio è risolta con un numero associato staticamente all'interfaccia del servizio (nomi unici); la fase statica di indirizzamento deve avere costi limitati ed accettabili durante il servizio e può essere:

- **esplicita**: il client deve raggiungere un server e lo fa mediante un multicast o un broadcast attendendo solo la prima risposta e non le altre;
- **implicita**: si fa uso di un name server che registra tutti i server e agisce su opportune tabelle di binding, ovvero di nomi, prevedendo funzioni di ricerca di nomi, registrazione, aggiornamento ed eliminazione.

In caso di binding dinamico, ogni chiamata richiede un collegamento dinamico; spesso, dopo un primo legame, si usa lo stesso binding ottenuto come se fosse statico per questioni di costo: il binding può quindi avvenire meno frequentemente delle chiamate stesse e in genere si usa lo stesso binding per molte richieste e chiamate allo stesso server.

Le RPC hanno portato a molti sistemi di nomi, detti binder, broker, name server, ecc., tutte entità di sistema per il binding dinamico. Un binder, per consentire agganci flessibili, deve fornire operazioni come:

- **Lookup** (servizio, versione, &servitore);
- **Register** (servizio, versione, servitore);
- **Unregister** (servizio, versione, servitore);

Il nome del server può essere dipendente dal nodo di residenza o meno; se dipendente, allora ogni variazione deve essere comunicata al binder. Il binding è attuato come servizio coordinato di più server: vengono usati binder multipli per limitare l'overhead ai singoli client o ai singoli nodi e inizialmente i client usano un broadcast per trovare il binder più conveniente.

8.9. RPC asincrone

Le RPC asincrone e non bloccanti permettono che il client non si blocchi ad aspettare il server; ne esistono di due tipi:

- **a bassa latenza**: tendono a mandare un messaggio di richiesta e a trascurare il risultato;
- **a throughput elevato**: tendono a differire l'invio delle richieste per raggrupparle in un unico messaggio di comunicazione.

A livello di implementazione si usano sia supporti UDP che TCP ottenendo semantiche diverse.

8.10. Proprietà delle RPC

Le proprietà delle RPC visibili all'utilizzatore sono:

- ci sono entità che si possono richiedere (operazioni o metodi di oggetti);
- la semantica di comunicazione (may-be, at-most-once, at-least-once);
- i modi di comunicare (sincrono o asincrono, bloccante o non bloccante);
- la durata massima e le eccezioni (ritrasmissioni e gestione degli errori).

Ciò che invece traspare all'utilizzatore è:

- la ricerca del servitore: uso di nomi con broker unico centralizzato o con broker multipli;
- la presentazione dei dati (linguaggio IDL ad hoc e generazione stub);
- il passaggio dei parametri (passaggio per valore o per riferimento);
- eventuali legami con le risorse del server (persistenza della memoria per le RPC, aggancio con il garbage collector).

9. Implementazione RPC di SUN

9.1. Introduzione

In questa implementazione RPC viene usata un'infrastruttura di supporto, sviluppata da SUN, inglobata nei processi impegnati nell'interazione. L'infrastruttura di supporto:

- scambia messaggi per consentire l'identificazione dei messaggi di chiamata e risposta e per consentire l'identificazione unica della procedura remota;
- gestisce l'eterogeneità dei dati scambiati, quindi esegue il marshalling, l'unmarshalling e la serializzazione degli argomenti;
- gestisce alcuni errori dovuti alla distribuzione, come l'implementazione errata, errori dell'utente, roll-over (ritorno indietro).

A fronte di possibilità di guasto, il client può controllare o meno il servizio con una semantica at-least-once; le operazioni eseguite dal server sono sequenziali e nel client sono sincrone bloccanti. Dato che il server è sequenziale, c'è possibilità di deadlock nel caso in cui un server RPC richieda, a sua volta, un servizio al programma chiamante. L'implementazione e il supporto RPC di SUN si basa sul modello ONC, *Open Network Computing*, una suite di prodotti che include:

- XDR, *External Data Representation*: rappresentazione e conversione dati;
- RPCGEN, *Remote Procedure Call Generator*: generazione degli stub di client e server;
- Portmapper: risoluzione dell'indirizzo del server;
- NFS, *Network File System*: file system distribuito di SUN.

9.2. Definizione del programma RPC

Le RPC sono basate su un contratto esplicito sulle informazioni scambiate, che consiste di due parti descrittive in linguaggio RPC:

1. definizioni di programmi RPC: specifiche del protocollo RPC per i servizi offerti, ovvero l'identificazione dei servizi e il tipo dei parametri;
2. definizione dei tipi di dati dei parametri: presente solo se il tipo di dato desiderato non è un tipo noto in RPC.

Queste due parti sono raggruppate in un unico file con estensione .x, cioè in formato XDR sorgente, che deve descrivere solo il contratto di interazione.

Durante la definizione del servizio remoto le RPC di SUN hanno i seguenti vincoli:

- ogni definizione di procedura ha un solo parametro d'ingresso e un solo parametro d'uscita;
- gli identificatori (nomi) usano lettere maiuscole;
- ogni procedura è associata ad un numero di procedura unico all'interno di un programma.

Il programmatore deve implementare:

- il programma client: va implementato il main(), la logica necessaria per il reperimento e il binding del servizio remoto (un file .c);
- il programma server: vanno implementate tutte le procedure, ovvero i servizi offerti (un file .c);

Lo sviluppatore non realizza il main() nel server e gli stub del client e del server vengono generati automaticamente.

9.3. Confronto tra sviluppo locale e sviluppo remoto

Il codice di un servizio è quasi identico alla versione locale, le differenze sono:

- sia l'argomento di ingresso che di uscita vengono passati per riferimento;
- il risultato punta ad una variabile statica (allocazione globale) per essere presente anche oltre la chiamata della procedura;

- il nome della procedura cambia leggermente (si aggiunge **__x__svc** alla fine, in cui x è il numero di versione);

Il client viene invocato con il nome dell'host remoto e il parametro che serve a scegliere il servizio. Nel main() del client bisogna creare il gestore del trasporto, ovvero una variabile di tipo **CLIENT*** che gestisce la comunicazione con il server e può usare sia UDP (default) che TCP. Questa variabile viene richiesta al portmapper attraverso l'invocazione della funzione **CLIENT* clnt_create(char* host, u_long n_prog, u_long n_vers, char* protocol)**.
Il client deve conoscere l'host remoto dove è in esecuzione il servizio e alcune informazioni per invocare il servizio stesso (programma, versione e nome della procedura).
I passi per sviluppare un programma RPC sono:

1. definire i servizi e i tipi di dati, se necessario;
2. generare in modo automatico gli stub del client e del server e, se necessario, le funzioni di conversione XDR tramite RPCGEN;
3. realizzare i programmi client e server, compilare tutti i file sorgente (programmi client e server, stub, file per la conversione dei dati) e fare il linking dei file oggetto;
4. pubblicare i servizi dal lato server: attivare il portmapper, se non attivo, e registrare i servizi presso il portmapper eseguendo il server;
5. reperire, dal lato client, l'end-point del server tramite il portmapper e creare il gestore di trasporto per l'interazione con il server.

9.4. Identificazione di RPC

Un messaggio RPC, per essere identificato globalmente, deve contenere il numero di programma, di versione e di procedura. Per i numeri di programma, che sono a 32 bit, SUN usa di default l'intervallo 0 – 1fffffffh per applicazioni di interesse comune; l'utente può definire numeri di programma nell'intervallo 20000000h – 3fffffffh per applicazioni a scopo di debug dei nuovi servizi e l'intervallo 40000000h – 5fffffffh è riservato alle applicazioni per generare dinamicamente numeri di programma; il resto è riservato per le estensioni. Per motivi di sicurezza l'identificazione del client presso il server, e viceversa, avviene sia in chiamata sia in risposta, quindi l'aggancio è dinamico.

9.5. Livelli di RPC

L'implementazione RPC di SUN ha tre livelli di uso:

- **alto**: è il livello utente; vengono forniti servizi RPC standard ai client, come ad esempio:
  - **rnusers()**: fornisce il numero di utenti di un nodo;
  - **rusers()**: fornisce informazioni sugli utenti di un nodo;
  - **rstat()**: ottiene dati sulle prestazioni verso un nodo;
  - **rwall()**: invia al nodo un messaggio;
  - **getmaster()**: ottiene il nome del nodo master per NIS;
  - **getrpcport()**: ottiene informazioni riguardo agli indirizzi TCP legati ai servizi RPC.
- **intermedio**: serve per definire ed utilizzare nuovi servizi RPC per procedure singole. Ci sono due primitive:
  - **callrpc()**, per il client: fa una chiamata esplicita al meccanismo RPC e prova l'esecuzione della procedura remota;
  - **registerrpc()**, per il server: associa un identificatore unico alla procedura remota implementata nell'applicazione. Quando viene utilizzata la procedura prima viene effettuata una ricerca in tabella della procedura e successivamente viene recapitata la richiesta al programma trovato.
- **basso**: serve per la gestione avanzata del protocollo RPC: le chiamate remote sono gestite tramite funzioni avanzate per ottenere la massima capacità espressiva; le chiamate

svc_register() e svc_run() possono essere implementate con funzioni di più basso livello. Per quanto riguarda il server sono presenti cinque funzioni:

- **svcudp_create()** e **svctcp_create()** per ottenere un gestore del trasporto UDP o TCP;
- **pmap_unset()** per distruggere eventuali precedenti registrazioni con lo stesso numero di programma e versione;
- **svc_register()** per associare una tripla (n_prog, n_vers, protocol) a una procedura di dispatching che implementa i vari servizi;
- **svc_run()** che rende il server un demone, quindi un ciclo infinito in attesa di chiamate.

Anche il client ha a disposizione cinque funzioni:

- **clntudp_create()** per creare un gestore del trasporto UDP. Tra gli argomenti di questa funzione c'è il valore del time-out fra le eventuali ritrasmissioni; se il numero di porta all'interno del socket address remoto vale 0, si lancia un'interrogazione al portmapper per ottenerlo;
- **clnttcp_create()** per creare un gestore del trasporto TCP. Non prevede un time-out e definisce la dimensione dei buffer di input e output; l'interrogazione iniziale causa una connessione e l'accettazione della connessione consente la RPC;
- **clnt_call()** per fare una chiamata alla procedura remota dopo che è stato creato il gestore del trasporto. La funzione specifica solo il gestore del trasporto e il numero della procedura;
- **clnt_perror()** per analizzare il risultato di clnt_call(). Stampa sullo standard error una stringa contenente un messaggio di errore;
- **clnt_destroy()** per deallocare lo spazio associato al gestore del trasporto del client, senza però chiudere la socket.

9.6. Omogeneità dei dati

Per far comunicare due nodi eterogenei ci sono due soluzioni:

- dotare ogni nodo di tutte le funzioni di conversione possibili per ogni rappresentazione dei dati: si ottengono performance elevate ma anche un elevato numero di funzioni di conversione, pari a n(n - 1);
- concordare un formato comune di rappresentazione dei dati: ogni nodo possiede le funzioni di conversione da/per questo formato; così facendo si ottengono performance minori ma il numero di funzioni di conversione, pari a 2n, è molto inferiore.

In questo caso il formato comune è XDR, che si occupa di svolgere la funzione di marshalling dei dati. Si colloca al livello 6 della pila OSI, il livello presentazione. XDR ha al suo interno procedure di conversione relative a tipi atomici predefiniti e tipi standard (costruttori riconosciuti). Gestisce anche strutture dati, infatti la primitiva callrpc() accetta solo un argomento ed un solo risultato, motivo per cui è necessario definire strutture dati per raggruppare gli argomenti. Per ogni informazione da trasmettere si devono effettuare due trasformazioni, una eseguita dal mittente e una dal destinatario; ogni nodo deve prevedere solamente le proprie funzionalità di trasformazione, quindi dal formato locale a quello standard e viceversa. In XDR sono presenti due tipi di definizione, già elencati nel paragrafo 9.2:

- definizione di tipi di dati: sono definizioni XDR per generare le definizioni in C e le relative funzioni per la conversione in XDR;
- definizione delle specifiche di protocollo RPC: sono definizioni di programmi RPC per il protocollo RPC (identificazione del servizio e parametri di chiamata).

9.7. Portmapper

Il portmapper è il server di nomi, lui stesso un server RPC. Identifica il numero di porta associato ad un qualsiasi programma ed esegue l'allocazione dinamica dei servizi sui nodi. Per limitare il numero di

porte riservate è presente un solo processo sulla porta 111. Il portmapper registra i servizi sul nodo e offre le seguenti procedure:

- inserimento di un servizio;
- eliminazione di un servizio;
- corrispondenza tra l'associazione astratta e la porta;
- intera lista delle corrispondenze;
- supporto all'esecuzione remota.

Di default utilizza il protocollo UDP e un limite è il fatto che il client per ogni chiamata interroga il portmapper e poi attua la richiesta.

9.8. Modalità asincrona batch

Di default un client RPC è sincrono con il server. Per renderlo asincrono è necessario:

- far usare il protocollo TCP al client;
- far specificare un time-out nullo al client, in modo tale da continuare immediatamente l'esecuzione;
- assicurarsi che il server non preveda una risposta.

Occorre quindi specificare un time-out nullo nella clnt_call() e nella clnt_control() per ogni chiamata; dato che il server non deve inviare alcuna risposta, nella procedura di risposta si deve dichiarare **xdr-void** come funzione XDR e passarle 0 come argomento. Si noti che il server ed il client basano la loro fiducia sul trasporto affidabile, quindi è importante usare TCP per evitare di perdere messaggi. Tutte le richieste di servizio del client vengono poste nel buffer TCP e gestite dal driver di trasporto senza bloccare il processo che le genera.

10. Sistemi di nomi

10.1. Introduzione

Spesso nei sistemi distribuiti ci si trova in presenza di molti sistemi di nomi che hanno molte proprietà e sono anche molto diversi. Alcune proprietà dei sistemi di nomi sono:

- generalità: c'è un'ampia varietà di nomi disponibili e trattabili;
- definizioni multiple della stessa entità: si può usare una varietà di nomi per lo stesso oggetto e grazie al mapping si può traslare tra questi;
- distribuibilità: si possono usare direttori partizionati e/o replicati;
- user-friendliness: i nomi sono facili per l'utente.

Un problema fondamentale nel distribuito è la necessità di ritrovare (ovvero identificare) le altre entità nel sistema: è un problema complesso e trovare soluzioni generali è difficile, infatti entità diverse eterogenee richiedono livelli diversi di nomi, quindi servono più sistemi di naming e più livelli di nomi nel sistema con contesti di visibilità: servono più funzioni di trasformazione da nome all'entità. I nomi possono essere identificati sia come stringhe di caratteri, quindi essere nomi esterni (nomi di utente) sia come numeri binari, quindi essere nomi interni (nomi di sistema).
Spesso si possono considerare alcuni livelli di nomi per il distribuito, ad esempio in tre possibili livelli:

- **nome logico esterno**: specifica a quale oggetto (entità) si riferisce e denota l'entità;
- **indirizzo fisico**: specifica dove l'oggetto risiede e lo riferisce dopo un binding;
- **route** (organizzazione per la raggiungibilità): specifica come raggiungere l'oggetto.

10.2. Sistemi di nomi

Gli spazi dei nomi più usati sono:

- **piatto**: con nessuna struttura, ma adatto per pochi utenti e poche entità;
- **partizionato**: serve per gerarchie e contesti (vedi DNS);
- **descrittivo**: con riferimento ad una struttura di oggetti caratterizzati da attribuiti per identificare l'entità corrispondente (vedi Directory X.500).

In questi scenari, spesso ci possono essere problemi nell'identificare nomi di gruppo, in quanto un nome di gruppo identifica una lista di nomi di entità e accade lo stesso quando si vuole fare un multicast rispetto a una comunicazione punto a punto.

10.3. Componenti di un sistema di nomi

In un servizio di nomi, consultato implicitamente o esplicitamente, i client sono:

- coloro che devono risolvere un nome per poter riferire una risorsa;
- le entità risorse (server rispetto ai client di prima, ossia che devono essere riferiti) che devono rendersi note al servizio e diventano client del name server.

A parte questi client e le loro richieste, il supporto deve anche considerare:

- la comunicazione dei client con il name server;
- il name server (anche più di uno);
- la gestione dei nomi veri e propri (coordinamento);
- la gestione delle tabelle e il coordinamento (spesso ottimizzato localmente).

Le comunicazioni dei client con il name server possono essere ottimizzate per le operazioni più frequenti:

- i client propongono la maggior parte del traffico;
- le risorse da registrare fanno operazioni più rare e producono traffico più limitato.

I name server devono fornire le operazioni per consentire la miglior operatività sugli oggetti interni, ossia le tabelle di corrispondenza modellate come tuple di attributi; ogni sistema di nomi decide il formato delle tuple ed il formato specifico delle operazioni. Le realizzazioni prevedono sia un unico server

sia agenti multipli; il servizio può anche essere centralizzato o molto più spesso distribuito e anche replicato (ad esempio DNS).
Nella realizzazione con molteplici name server, il servizio prevede una comunicazione tra loro usando messaggi singoli, datagrammi, connessioni, invocazioni di alto livello come RPC. Il traffico tra i diversi name server deve essere supportato mentre si continua a fornire il servizio e il coordinamento dei server deve essere minimizzato in termini di tempo e uso delle risorse, tenendo conto anche delle proprietà che si vogliono garantire:

- in uno spazio piatto, occorre una partizione dello spazio dei nomi per limitare la coordinazione;
- in uno spazio partizionato, i nomi sono usati dall'autorità preposta senza coordinamento.

La gestione delle tabelle e il coordinamento creano problemi di consistenza e affidabilità. Nella gestione dei nomi sono fondamentali due decisioni:

- la distribuzione dei nomi: i nomi sono mantenuti in oggetti che ne hanno la responsabilità o autorità con un partizionamento tra i server responsabili; la gestione e il mantenimento sono divisi in modo algoritmico, sintattico o basato su attributi;
- la risoluzione dei nomi: le richieste dal client devono fluire fino al server che può dare una risposta; questo processo, di risoluzione, prevede alcune fasi (non sempre presenti):
  - trovare l'autorità corretta;
  - verificare le autorizzazioni all'operazione;
  - eseguire l'operazione.

  Ogni nodo specifica i name server noti e tende a limitare, se possibile, le comunicazioni tra i server. Strategie usuali per limitare i costi sono:
  - politiche di caching;
  - politiche di routing tra server;
  - creazione di contesti o vicinati tra i server;
  - propagazione di conoscenza tra i vicini.

Nei sistemi come DNS si distribuisce e si risolve il nome in contesto locale e si ricorre ad altri contesti solo se necessario: in tal caso si utilizzano risoluzioni ricorsive o iterative; se possibile, le strategie di coordinamento tra i server devono essere a basso costo.

10.4. Directory X.500

È un servizio standard di directory e di nomi con realizzazione partizionata, decentralizzata e disponibile 24/7. È definita come una collezione di sistemi aperti che cooperano per mantenere un database logico di informazioni sugli oggetti del mondo reale; gli utenti della directory possono leggere o modificare l'informazione, o parte di essa, solo se hanno i privilegi necessari. La base è l'insieme delle informazioni che caratterizzano la struttura di directory, organizzate in un albero logico detto **DIT**, *Directory Information Tree*, che forma il **DIB**, *Directory Information Base*. La novità sta nell'organizzazione basata sui contenuti e sulle ricerche (operazioni di lettura) che si possono fare, in modo flessibile, per singole entità e anche per attributi, ritrovando gruppi di elementi (nodi) anche molto numerosi. Ogni entry, o nodo, si trova attraverso diverse notazioni:

- **DN**, *Distinguished Name*: identifica univocamente l'oggetto all'interno del DIT;
- **RDN**, *Relative Distinguished Name*: definisce univocamente un oggetto all'interno di un contesto.

Le ricerche possono essere fatte in modo globale o contestuale per uno specifico DN ma anche per contenuto dei nodi, in base agli attributi. Usando i filtri, strumenti molto potenti come capacità espressiva, si possono sfruttare condizioni logiche, espressioni regolari e condizioni aritmetiche sugli attributi. La ricerca sulla directory X.500 avviene attraverso agenti:

- **DUA**, *Directory User Agent*: è il tramite per fare le richieste;
- **DSA**, *Directory System Agent*: mantiene le informazioni di contesto;
- **DSP**, *Directory System Protocol*: permette lo scambio di informazioni tra DSA;
- **DAP**, *Directory Access Protocol*: è il protocollo di accesso alla directory.

Dopo la connessione, si fanno operazioni di lettura, confronto, ricerca, lista delle entità con tecniche di ricerca ricorsive e iterative. È anche presente **LDAP**, *Lightweight Directory Access Protocol*, un protocollo limitato, compatibile con Internet, usato per infrastrutture di verifica certificati.
A differenza di un database, in una directory:

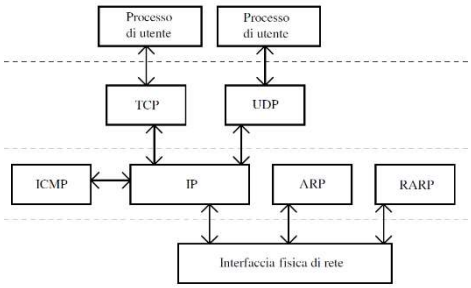
- si associano attributi anche diversi a singoli oggetti;
- gli oggetti sono indipendenti tra di loro e possono essere diversi;
- si considera la relazione di contenimento alla base dell'organizzazione;
- si possono avere proprietà di accesso differenziate per i singoli oggetti;
- si ottimizza considerando un numero elevato di letture e poche scritture.

10.5. Protocolli di directory e protocolli di discovery

Considerando che un'entità può avere sia attributi con lente variazioni sia attributi con rapide variazioni, le directory sono soluzioni di nomi globali, servizi completi e complessi con costo elevato delle operazioni; per questo esistono anche le discovery, soluzioni di nomi locali, ovvero servizi essenziali con funzioni limitate dal costo limitato, adatti a variazioni rapide.
Un servizio di directory garantisce le proprietà di QoS, ossia di replicazione, sicurezza, gestione multi-server, tutto il supporto per memorizzare le informazioni organizzate prevedendo molti accessi in lettura e poche variazioni: alcuni esempi sono UPnP, *Universal Plug And Play*, standard per architetture Microsoft e Salutation, *Service Location Protocol*, definito in RFC 2165.
I protocolli di discovery sono usati per computazione distribuita e cooperativa in ambito locale: ad esempio un'unità deve ritrovarne altre, in modo veloce ed economico, o sono previsti azioni come il broadcast e solleciti periodici. Un esempio è JINI, un protocollo Java per il discovery di appliance.

11. TCP/IP: protocolli e scenari d'uso

11.1. Introduzione



Gli standard possono nascere da comitati, come ad esempio OSI, o anche da esigenze di uso e con obiettivo di realizzazione immediata, come la suite TCP/IP. Partendo dal basso, il livello 2 è formato da ICMP, un protocollo per lo scambio di messaggi di controllo, IP, un protocollo che permette di scambiare datagrammi senza garanzia di consegna tra vicini, ARP e RARP, due protocolli che permettono l'interazione con il livello fisico e i nomi. Salendo al livello 3 è presente il protocollo TCP, che permette di scambiare informazioni tramite un flusso di byte, bidirezionale a canale virtuale, con controllo di flusso, dati affidabili e non duplicati e il protocollo UDP, che permette lo scambio di messaggi end-to-end senza garanzie o controlli. Nella suite TCP/IP non sono consentiti broadcast a livello globale in quanto, data la dimensione del sistema, il costo sarebbe inaccettabile; il broadcast è quindi permesso solo nell'ambito di una rete locale. Un broadcast può essere:

- **limitato**: per tutti gli host della rete locale. L'indirizzo ha tutti i 32 bit a 1 ed è solo per la rete locale, quindi non viene fatto passare da una rete all'altra;
- **diretto**: viene fatto per tutti gli host di una rete specifica. Tutti i bit di *hostid* sono a 1 e viene trasmesso in Internet fin quando non arriva a destinazione.

È anche possibile effettuare multicast sfruttando gli indirizzi IP di classe D: tutti gli host che si sono registrati possono ricevere messaggi e possono mandare messaggi al gruppo di multicast; tuttavia, i messaggi multicast sono un'implementazione recente e sono soltanto in via sperimentale per ora.

11.2. ARP – *Address Resolution Protocol*

Una macchina ha sia un indirizzo IP sia un indirizzo fisico MAC. Quando una macchina deve comunicare con un'altra macchina, sa il suo indirizzo IP ma è necessario anche l'indirizzo fisico MAC per identificare univocamente quella macchina; l'indirizzo MAC potrebbe essere ottenuto mediante un mappaggio diretto, estraendo l'indirizzo MAC dall'indirizzo IP, ma violerebbe l'indipendenza dei livelli e inoltre i nomi sono svincolati: è quindi necessario un protocollo dinamico che permetta di ritrovare l'indirizzo MAC a partire dall'indirizzo IP (e viceversa). Mediante ARP, un protocollo locale, semplice ed efficiente basato su broadcast, è possibile ricercare l'indirizzo MAC di un nodo a partire dal suo indirizzo IP; nel caso di una macchina A che vuole comunicare con una macchina B:

- la macchina A ha l'indirizzo IP della macchina B; la macchina A invia un pacchetto broadcast in cui chiede qual è l'indirizzo MAC della macchina B;
- tutti gli host ricevono questo pacchetto e solo quello che riconosce il suo indirizzo IP risponde alla macchina A inviando il proprio indirizzo MAC.

Per ottimizzare il protocollo ed evitare di avere dei costi di esercizio troppo alti, ARP adopera una memoria cache per mantenere le associazioni {indirizzo IP – indirizzo MAC} già usate e questa viene consultata ogni volta prima di effettuare un broadcast.

11.3. RARP – *Reverse Address Resolution Protocol*

Il protocollo RARP è usato per risalire all'indirizzo IP a partire da un indirizzo fisico MAC. Questo protocollo permette anche ad un host di venire a conoscenza del proprio indirizzo IP all'accensione, chiedendolo in modalità broadcast agli altri host connessi alla rete. In genere, l'indirizzo IP si mantiene sul disco e lì viene trovato dal sistema operativo; per le macchine senza disco invece, l'indirizzo IP viene ottenuto richiedendolo ad un server di rete che contiene tutti gli indirizzi IP di una rete. Spesso possono

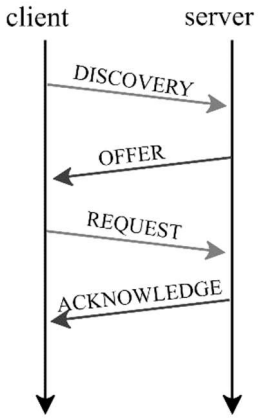
esserci server multipli per RARP ma si rischia di sovraccaricare il sistema se cercano di rispondere contemporaneamente ad una richiesta: per evitare queste situazioni ci sono due possibili modelli:

- avere un server primario che è l'unico a rispondere ad una prima richiesta e gli altri server rispondono solo se arriva una seconda richiesta;
- prevedere che il server primario risponda immediatamente ad una richiesta e gli altri con un ritardo random, in modo da avere una bassa probabilità che rispondano contemporaneamente.

Tuttavia, a differenza di ARP, RARP è un protocollo ormai deprecato e si usa il più moderno DHCP.

11.4. DHCP – *Dynamic Host Configuration Protocol*

DHCP è un protocollo per l'attribuzione dinamica di indirizzi IP che permette di risparmiare indirizzi, attribuendoli solo al bisogno e non in modo statico. In particolare, DHCP risulta utile per far risparmiare ai provider gli indirizzi IP e far sì che vengano assegnati solo al bisogno e non in modo permanente. Il protocollo prevede un client e un server che negoziano un'offerta a più fasi in cui l'iniziativa è del client. In ordine:



1. un client che vuole ottenere un indirizzo IP fa un broadcast della richiesta di discovery inviando DHCPDISCOVER;
2. i server gli inviano un'offerta DHCPOFFER con parametri di scelta;
3. il client sceglie un'offerta tramite DHCPREQUEST;
4. viene confermata l'offerta dal server con l'invio di DHCPACK al client;
5. il client invia il rilascio dell'offerta con DHCPRELEASE o invia DHCPLEASE per mantenere l'IP.

Le moderne organizzazioni tendono a usare il protocollo DHCP per gestire molti host di un'organizzazione evitando il set-up manuale o statico e per ragioni di sicurezza. Al contratto viene associata una durata: se durante l'intervallo non si usa, il server può riassegnare l'indirizzo. Il lease permette di confermare l'uso senza rieseguire il protocollo.

11.5. NAT – *Network Address Translation*

NAT è un protocollo usato per traslare gli indirizzi IP di una rete privata in indirizzi IP globali, cioè di una rete aperta. Grazie a NAT si può evitare di attribuire un indirizzo IP pubblico a ciascun host di una rete intranet privata: ogni host di una rete intranet privata utilizza un indirizzo IP privato, ovvero un particolare indirizzo IP che lo identifica soltanto nella rete intranet e grazie alle tabelle di corrispondenza presenti nel router (che integrano NAT) è possibile traslare questi indirizzi in indirizzi IP pubblici. In questo modo molti indirizzi interni sono mappati in un unico indirizzo esterno, permettendo di risparmiare IP pubblici.

11.6. ICMP – *Internet Control Message Protocol*

ICMP è un protocollo di gestione e controllo degli indirizzi IP che serve a migliorare la qualità best-effort. Consente di inviare messaggi di controllo o di errore al nodo sorgente del messaggio (e solo a questo). È anche usato per coordinare le entità di livello IP; rappresenta un mezzo per rendere note condizioni anomale a chi ha mandato datagrammi usando l'indirizzo IP. I nodi intermedi non sono informati dei problemi ma soltanto il nodo sorgente può provvedere a correggere. Per risalire al mittente di un messaggio si utilizza il normale routing IP: i messaggi ICMP, infatti, sono contenuti all'interno di un datagramma IP; non hanno priorità e possono essere persi, ma in questo caso non si inviano ulteriori messaggi ICMP per evitare congestioni. Un messaggio ICMP contiene sempre l'header e i primi 64 bit dell'area dati del datagramma che ha causato il problema; l'header è formato da tre campi:

- **type**: specifica il formato del messaggio;
- **code**: fornisce un'ulteriore qualificazione del messaggio;
- **checksum**: un campo per controllare la correttezza del messaggio.

11.7. UDP – *User Datagram Protocol*

UDP è un protocollo di trasporto best-effort e a basso costo. Per distinguere tra più processi in esecuzione su un nodo connesso alla rete, si identificano i processi con un nome, formato dall'indirizzo IP e dal numero di porta. Il servizio fornito da UDP è unreliable e connectionless: i datagrammi possono essere persi, duplicati, pesantemente ritardati o consegnati fuori ordine ed è il programma applicativo che usa UDP che deve trattare i problemi. I messaggi UDP sono composti da header e area dati, con l'header formato da porte, lunghezza messaggio e checksum. Un messaggio UDP è totalmente contenuto nell'area dati di un datagramma IP, senza nessuna frammentazione: banda e overhead del protocollo sono molto limitati. Le porte possono essere assegnate in modo statico, in cui un'autorità centrale pre-assegna i numeri di porta universalmente validi, o in modo dinamico, in cui l'assegnamento è su necessità con un binding dinamico, ovvero i numeri di porta non sono assegnati a priori ma su richiesta. In UDP e TCP si adotta una soluzione ibrida: alcuni numeri di porta sono noti a priori, detti well known port, mentre gli altri sono assegnati dinamicamente.

11.8. ARQ - *Automatic Repeat Request* e Continuous Requests

Per ottenere qualità del servizio occorre trovare un compromesso fra affidabilità, attesa dell'avvenuta ricezione di un messaggio e asincronismo, ovvero non attendere troppo tempo. ARQ è una strategia di controllo di errore che svolge il compito di rivelare un errore ma senza correggerlo; i pacchetti corrotti vengono scartati e viene richiesta la loro ritrasmissione. Affinché il sistema sia capace di riconoscere i messaggi corrotti è necessario che questi vengano inizialmente codificati da un codificatore e dopo la trasmissione un decodificatore li legge. Ci sono tre possibili modi per fare ciò:

- **Stop-and-wait**: il mittente invia un messaggio e attende dal destinatario un messaggio di conferma positivo, detto ACK; se scade il tempo di attesa per uno dei messaggi, il mittente deve rispedire il pacchetto e il destinatario si occuperà di scartare eventuali messaggi replicati;
- **Go-Back-N**: il mittente ha a disposizione un buffer dove memorizza N pacchetti da spedire e man mano che riceve la conferma ACK svuota il buffer e lo riempie con nuovi pacchetti; nell'eventualità di pacchetti persi o danneggiati e scartati avviene la rispedizione del blocco di pacchetti interessati;
- **Selective Repeat**: in questo caso anche il destinatario dispone di un buffer dove memorizzare i pacchetti ricevuti: quando i pacchetti interessati vengono correttamente ricevuti, entrambi i buffer vengono svuotati.

Per superare i problemi di sincronia si adopera il protocollo **Continuous Requests**: si manda un certo numero di messaggi, pari alla dimensione del buffer, in modo ripetuto. Il mittente mantiene un buffer per i messaggi e si blocca solo quando è pieno; quando arriva un messaggio di ACK il messaggio è tolto dal buffer e il ricevente lo passa al livello superiore in ordine. È un protocollo più complesso di ARQ e le conferme sono con overhead; per la comunicazione full-duplex gli ACK sono mandati in piggybacking sul traffico opposto. In caso di errore o di messaggi saltati ci sono due strategie attuabili:

- **Selective Retransmission**: si attende l'esito dei messaggi tenendo conto degli ACK ricevuti e anche degli ACK negativi (dovuti al time-out del ricevente) e si ritrasmettono quelli persi;
- **Go-Back-N**: si attende l'ACK di conferma e si ritrasmettono tutti i messaggi anche se solo uno ha problemi.

11.9. TCP – *Transmission Control Protocol*

TCP è un protocollo per la trasmissione dei dati che permette la connessione end-to-end tra processi di nodi distinti. Un end-point è definito come una coppia di interi {indirizzo IP, porta} e una connessione TCP è identificata univocamente da una quadrupla {indirizzo IP 1, porta 1, indirizzo IP 2, porta 2}. Può esistere una sola connessione per quadrupla ma possono esserci, ad esempio, più connessioni che utilizzano una stessa porta, a patto che cambi uno degli altri tre parametri. Il servizio di trasmissione di TCP è uno stream full duplex affidabile. La connessione end-to-end garantisce che il messaggio passi, dalla memoria del mittente alla memoria del destinatario, con successo e che si mantenga il flusso; il flusso, è costituito da dati in ordine preciso e non alterabile. La banda è limitata unicamente per i dati urgenti, che possono essere al più 1 byte. Nell'header TCP, costituito da almeno 20 byte, sono presenti 5 parole:

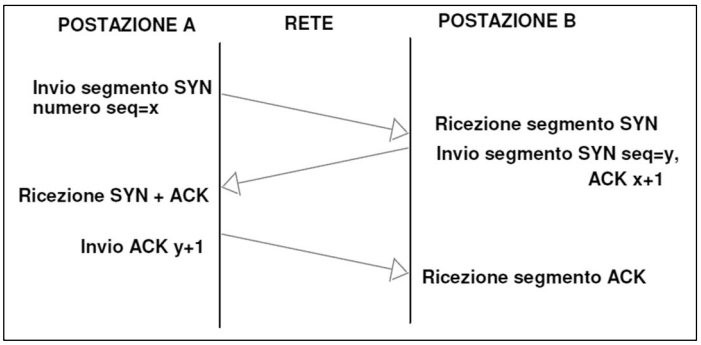
- le porte del mittente e del destinatario;
- il numero di sequenza;
- il numero di ACK;
- la lunghezza dell'header, della finestra di ricezione del mittente e il code bit;
- un puntatore urgente e il checksum di controllo.

Possono esserci anche delle eventuali opzioni aggiuntive, 6, indicate nel campo code bit:

- **URG**: se il bit è a uno, è presente un dato urgente nel flusso e questo viene segnalato da ogni header di segmento inviato al ricevente;
- **ACK**: se il bit è a uno, nel campo relativo c'è un ACK significativo;
- **PUSH**: se il bit è a uno, si richiede l'invio immediato del segmento senza bufferizzarlo;
- **RST**: se il bit è a uno, si richiede di riavviare la connessione in quanto sono stati riscontrati problemi;
- **SYN**: se il bit è a uno, si vuole stabilire la connessione;
- **FIN**: se il bit è a uno, si vuole iniziare la fase di chiusura della connessione dal lato mittente.

TCP tende a spezzare i messaggi in segmenti di dimensione variabile, in modo che non siano né troppo corti, causando un grosso overhead di trasmissione, né troppo lunghi, causando frammentazione a livello di IP e possibili perdite. Si usa Continuous Request per efficienza ed affidabilità: i messaggi prevedono ACK, ed essendoci traffico in entrambi i sensi, gli ACK sono inseriti sul traffico in direzione opposta (piggybacking). L'arrivo dell'ACK di un messaggio implica che sono arrivati anche i messaggi precedenti. Viene usato un buffer, detto **sliding window**, di dimensione decisa dal ricevente e stabilita per ogni invio di segmento. In caso di non ricezione di un segmento si usa Go-Back-N; i parametri decisi dal protocollo e non visibili all'esterno sono: dopo quanto tempo si ritrasmette, quante volte si esegue la ritrasmissione e come si frammentano i segmenti.

11.10. TCP: fase iniziale



Per stabilire la connessione TCP il mittente attua un protocollo, detto **three-way handshake**, per realizzare la connessione tra i due driver di protocollo dei due nodi. Sono tre fasi di comunicazione per il coordinamento iniziale tra mittente A, che gioca un ruolo attivo, e il ricevente B, nel ruolo passivo, con valori random per etichettare il byte d'inizio del flusso di comunicazione nei due versi; ogni byte è collocato nel suo stream. Il three-way handshake avviene nel seguente modo:

1. A invia a B il segmento con SYN e richiede la connessione (SYN si trova nell'header ed è presente un valore iniziale del flusso, X, scelto da A);
2. B riceve il segmento con SYN e ne invia uno identico ad A con ACK, in questo caso con un valore iniziale del flusso Y scelto da B per il suo verso;
3. A riceve il segmento SYN e ACK e conferma la ricezione a B attraverso un ACK a sua volta.

Queste tre fasi sono necessarie per ottenere la semantica at-most-once. La negoziazione a tre fasi serve per stabilire se entrambi i nodi sono disponibili alla connessione. Sempre nella comunicazione iniziale si effettua anche un coordinamento di vari parametri, come ad esempio i numeri di porta disponibili, il tempo di trasmissione, il tempo di risposta, la dimensione del buffer di ricezione.

11.11. TCP: fase di comunicazione

A regime, si fanno continui aggiornamenti dei valori in base alla situazione corrente rilevata. Dopo il calcolo iniziale del **time-out**, lo si ricalcola per ogni segmento in base al tempo di percorrenza medio di andata e ritorno. Si sono diffusi diversi algoritmi di calcolo del time-out, sempre tenendo conto di criteri di minima intrusione e di massima efficienza. Il time-out tiene conto della storia pregressa e del valore ricavato correntemente con pesi diversi, per non essere né troppo reattivo né troppo conservativo. Il time-out principale viene calcolato multiplo di 100, 200 o 500 ms, in modo da dover gestire una granularità limitata. Il time-out principale è la base di molti parametri temporizzati per la connessione:

- il ricevente differisce i messaggi corti di ACK in modo da sfruttare il piggybacking sul traffico, usando un time-out per limitare il ritardo massimo: dopo il time-out si invia un segmento ad-hoc di controllo;
- il mittente differisce i byte applicativi e li mantiene fino ad aver raggiunto un segmento di dimensione media: dopo un certo time-out, si invia il messaggio corto in ogni caso per non incorrere in troppo ritardo;
- entrambi gli end-point ritardano i messaggi all'applicazione fino ad avere segmenti medi: se il bit PUSH è a 1 si invia e riceve rapidamente;
- entrambi gli end-point, in caso non ci sia traffico, mandano messaggi di verifica del pari in modo da sapere la situazione corrente della connessione: dopo un intervallo lungo, si invia un segmento di controllo.

Il **controllo di flusso** è fondamentale in Internet, essendoci connessioni tra macchine molto diverse tra loro. Meccanismi fondamentali di coordinamento sono la sliding window e la dimensione preferenziale dei segmenti da inviare; per quanto riguarda la sliding window:

- la sua dimensione viene inviata per ogni segmento e comunica al pari quali siano le esigenze di memoria della connessione;
- se la sua dimensione è pari a 0 significa che non va inviato alcun segmento;
- ogni pari comunica all'altro la propria situazione con la finestra.

Per quanto riguarda la dimensione preferenziale dei segmenti da inviare, invece, i dati non vengono inviati fin quando non si ha un segmento di dimensione conveniente da inviare, con lo scopo di evitare la trasmissione di messaggi corti; un algoritmo usato per evitare di inviare messaggi corti è l'algoritmo di Nagle. Le applicazioni possono anche cercare di superare la trasparenza di TCP:

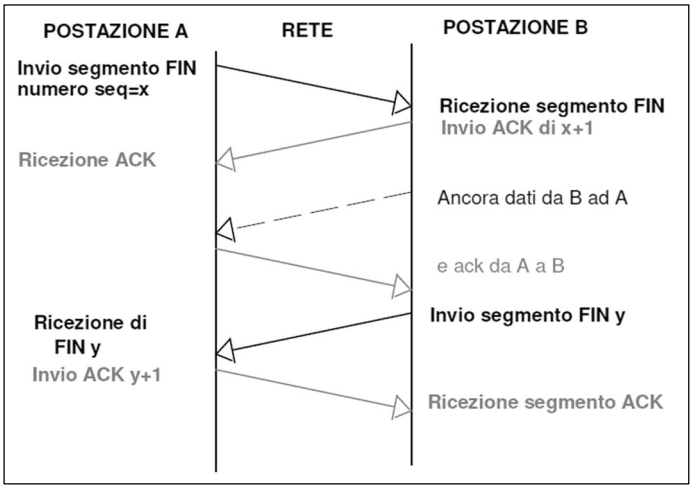
- inviando un segmento con il code bit PUSH a 1: il segmento viene inviato immediatamente e portato all'applicazione il prima possibile;
- inviando un segmento con il code bit URG a 1: si segnala la posizione, nel flusso, del byte urgente e il ricevente deve consumare i dati per arrivare quanto prima al byte urgente.

A regime, ogni segmento inviato produce coordinamento con il pari attraverso l'header del segmento stesso. Ogni segmento invia sempre informazioni di controllo al pari, come la propria posizione nel flusso, la posizione nel flusso ricevuto con ACK, la finestra di accettazione corrente nella propria direzione. Il ricevente adegua i propri parametri, come la dimensione della sliding window di cui è mittente e i time-

out, misurati e riadeguati. La congestione è critica per TCP: non si riescono più a consegnare dati in tempi utili rispetto all'operatività corrente. La congestione può essere sia dipendente dai soli end-point della connessione che dall'intera rete; tutti i router sono con buffer pieni e nessuno scambio può avvenire, fino alla de-congestione: in questo caso si può cercare di limitare i danni locali o di evitarli a priori. Quando si ha congestione il time-out scatta in modo ripetuto: si assume che il pari non sia raggiungibile e che la congestione sia in atto. Per il recovery, si devono attuare azioni locali per evitare di aggravare il problema e di scongiurare la congestione: in modo unilaterale, il mittente dimezza la finestra di invio e raddoppia il time-out; al termine della congestione, per ritornare ad una situazione di regime, si riparte con un transitorio con finestra piccola, detto **slow start**. Lo slow start è anche la politica usata per evitare una potenziale congestione iniziale: le variazioni vengono fatte in modo dolce, infatti inviando subito tutto il flusso si causerebbero probabilmente dei transitori di congestione sui router intermedi. La connessione TCP adotta meccanismi di controllo della congestione, senza usare solo la finestra del ricevente: si lavora con una finestra ulteriore, variabile in dimensione, detta di congestione. Si considera anche un ulteriore valore variabile, detto *ssthreshold* o soglia di slow start. Oltre allo slow start esiste anche la poltiica di **congestion avoidance**. In caso di congestione presunta, si ridimensiona tutto e si riparte in modo esplorativo dopo un time-out ripetuto, inteso come indicatore di congestione (o anche uno solo, se viene rilevata la congestione). Se viene rilevata la congestione, si riparte con una congestion window iniziale e si considera una nuova soglia di congestione limitata, iniziando con uno slow start.
A titolo informativo, alcune strategie tipiche in TCP sono:

- **ricalcolo del time-out in modo dinamico**: il time-out corrente viene tarato rispetto a quanto calcolato come media con la stima del time-out precedente;
- **exponential backoff**: in caso di ritrasmissione, il time-out raddoppia continuamente fino ad un tempo massimo, poi si chiude la connessione;
- **silly window**: per evitare di lavorare un byte alla volta, non si annunciano finestre di dimensione troppo piccole a parte la finestra chiusa;
- **limiti al time-wait**: serve per limitare la durata delle risorse per la connessione;
- **long fat pipes**: per mantenere piene le pipe a banda elevata, fornendo indicazioni di buffer superiori a quelli di utente e bufferizzando a livello di supporto.

11.12.TCP: fase finale



La chiusura di una connessione TCP avviene in 4 fasi; è monodirezionale nel verso di output, ovvero definitiva per un solo verso (quello di autorità) senza perdita dei messaggi in trasferimento e di quelli in arrivo. Se ad esempio A chiude nel suo verso di uscita, A comunica a TCP di non avere ulteriori dati e chiude e TCP chiude la comunicazione solo nel verso da A a B; si noti che su un canale chiuso è comunque possibile far passare messaggi di ACK. Nella chiusura A invia il segmento FIN in ordine dopo l'invio dei dati precedenti: TCP aspetta a chiudere la trasmissione, ma invia da A a B solo ACK; intanto, B può continuare ad inviare dati ad A. Al termine del traffico applicativo da B ad A, B invia ad A il segmento FIN che informa della disponibilità a chiudere la connessione. L'ultimo passo conferma da A a B la ricezione del segmento FIN e la chiusura totale della connessione.